1

EEBus SPINE Technical Specification

Protocol Specification

Version 1.1.1

Cologne, 2018-12-17

EEBus Initiative e.V.
Butzweilerhof Allee 4
50829 Cologne
GERMANY

Rue d'Arlon 25
1050 Brussels
BELGIUM

Phone: +49 221 / 47 44 12 - 20
Fax: +49 221 / 47 44 12 - 1822

info@eebus.org
www.eebus.org

District court: Cologne
VR 17275

The EEBus concept was developed as part of E-Energy.

Supported by: Federal Ministry of Economics and Technology on the basis of a decision by the German Bundestag

Table of contents

List of figures

List of tables

1 Introduction

This document describes the SPINE protocol to realize interactions between SPINE devices. It makes use of the underlying functionality of the SPINE data model, described in the document [ResourceSpecification].

A developer of a SPINE device will get details about structure, sequences and tables concerning the general format of a SPINE Datagram, useful rules and descriptions of the functional commissioning (especially for detailed discovery, binding and subscription mechanism) between SPINE devices and functionalities during runtime.

1.1 References

1.1.1 EEBUS SPINE documents

[Introduction] EEBus_SPINE_TR_Introduction.pdf

[ResourceSpecification] EEBus_SPINE_TS_ResourceSpecification.pdf

[TechnologyMappings] EEBus_SPINE_TS_TechnologyMappings.pdf

[DataModelXSDs] EEBus_SPINE_TS_ActuatorLevel.xsd, ..., EEBus_SPINE_TS_Version.xsd

1.1.2 Other EEBUS documents

[SHIPSpecification] SHIP_Specification_V1.0.0.pdf

1.1.3 Websites

[EEBus] http://www.eebus.org *Official EEBus Initiative e.V. website.*

[IANA PEN] http://pen.iana.org/pen/PenApplication.page *Website for requesting an IANA PEN.*

[W3C] http://www.w3.org/ *Official World Wide Web Consortium website.*

[W3Schools] http://www.w3schools.com/ *Tutorials and examples for XML and others.*

1.1.4 Normative References

[RFC2119] IETF RFC 2119: 1997, Key words for use in RFCs to indicate requirement levels Please see section 1.3.1 for details.

1.2 Terms and definitions

Binding

Concept for connecting functionally matching features.

(Standard or Complex) **Class**
Set of SPINE functions used to describe a specific functionality. A class can be considered as a topic where functions are defined for. For example, the SPINE class “Measurement” is a collection of SPINE functions that are used to describe measurement values.

Classifier
Specifies whether a message serves to read, reply, write, etc.

Client
Role that specifies that a node uses data from a "server" or can change it.

Command
The functional part of a Message.

Complex Class
SPINE class that is build up by parts of SPINE standard classes and combines them in a new, ordered way.

(SPINE) **Data model**
Definition of the possible data that can be used for SPINE communications. Defined as XSD.

Device (specific node)
SPINE node that can include a set of entities. It has a “deviceType”. With regards to the hierarchy of SPINE nodes a device is a root node for all functionalities offered by a device.

“device” (address information)
SPINE address part for the (physical) device.

DeviceType
Specific type of physical device (e.g. "WashingMachine", "HeatPump", "FridgeFreezer", etc.).

Discovery
Process of finding appropriate partners for communication. Dependent on the context this can be either finding other devices or examination of a device’s potential functionalities.

Element
Item (or “attribute”) of a SPINE function. Holds one information (e.g. "timestamp", "value", etc.) or contains further sub-elements.

Entity (specific node)
SPINE node that can include a set of (child) entities or features. It has an “entityType”. With regards to the hierarchy of SPINE nodes an entity is a child element of a device.

“entity” (address information)
SPINE address part for the (logical) entity.

EntityType
Specific type of logical device (e.g. "Freezer" is one logical part of a physical device "FridgeFreezer").

Feature (specific node)
SPINE node that can include a set of functions (of a class). It has a “featureType”. With regards to the hierarchy of SPINE nodes a feature is a child element of an entity.

“feature” (address information)
SPINE address part for one feature.

FeatureType
Defines optional or mandatory rules and a general behaviour of the underlying Class (standard or complex).

(SPINE) Function
A (SPINE) function is the smallest structure to model “actual data” (“functional data”). I.e. functions usually consist of child elements that each hold an information (e.g. "timestamp", "value", etc.). Information between communication partners is exchanged via the exchange of a function (as part of a so-called “payload”).

Header
SPINE Header, including elements for addressing, unique identification of messages, timestamp, etc.

Message
One SPINE transfer from a sender to a receiver.

(XML) **Namespace**
XML namespaces provide a simple method for qualifying element and attribute names used in XML documents by associating them with namespaces identified by URI references (source: www.w3.org).

Node
Common term for a SPINE instance that has a SPINE address. Dependent on the situation a node can be either a device or an entity (of a specific device) or a feature (of a specific device-entity).

Official EEBUS use cases

Use Cases that are released as official EEBUS Use Case specification through the EEBus Initiative e.V.

Payload
SPINE Payload, containing the functional SPINE data.

Role
Each Feature has a functional role, usually either “server” (data owner) or “client”. For some special features (NodeManagement, e.g.) the role “special” is defined.

Scope (Type)
Some feature types define scope types for identifying specific functionalities unambiguously (e.g. *outsideAirTemperature*).

Server
Role that specifies that a node offers own data to be read or written by a node with role client. A server can notify its data to other nodes (with role client).

SPINE
Smart **P**remises **I**nteroperable **N**eutral-message **E**xchange

Standard Class
All basic/standard functions are defined in standard classes. Functions of standard classes follow very simple patterns and do not have deeply nested data structures.

Subscription
Enables the receiving of messages of interest from another device without polling it.

Use case
Textual description of a re-usable functionality consisting of one or more messages of one or more participating actors. May be visualized with a sequence diagram. E.g. "A CEM shifts the energy usage of a washing machine."

User story
Complete (but specific) business case described from the perspective of a user. Can be separated into several use cases. E.g. "The user wants to get the laundry done by 8:00pm."

XML (Extensible Markup Language)
Human- and machine-readable markup language containing data. Used to model SPINE messages.

XSD (XML Schema Definition)
Definition format for XMLs, written in XML. Specifies how a well-formed XML (in regards to this XSD) can be built. The SPINE data model is defined in XSD and supplementary documents (as not every rule can be specified with XSD only). Other formats than XML can be derived from an XSD, too (e.g. JSON).

1.3 How to read this document

1.3.1 Used requirement keywords

The following keywords are used:

- SHALL
- SHALL NOT
- SHOULD
- SHOULD NOT
- MAY

They apply only, if written in capital letters!

For the meaning of the keywords, please refer to [RFC2119].

2 General notations

2.1 Data model specialization: From “generic XSD models” to “adjusted models” in “tables” and feature types

The SPINE concept aims to permit reuse of definitions as much as possible, esp. for the data model. This approach supports re-using the same basic data structure for different functionalities. In a first step, required data structures are modelled (using XSD, see esp. [ResourceSpecification], Annex A) with the following basic idea in mind: Define all elements which may be needed for specific purposes, but declare them being optional. This permits using the same data structure for different functionalities – esp. tagged by so-called "feature types" – where the feature types have different requirements regarding the presence of the elements. I.e. esp. each feature type can then define if a specific element must be present or must be absent or may remain optional. The subsequent sections briefly explain how these kinds of specializations are described in this specification.

Please note that feature types are primarily defined in [ResourceSpecification]. However, the design concept described above applies as well for functionalities defined by the specification at hand.

2.1.1 Element presence indications

The following abbreviations on the presence of elements or the support of features are used:

1. M = mandatory
2. O = optional
3. NV = Not valid
4. C = “choice”, i.e. a presence or support depends also on the selection from multiple possibilities

These symbols are primarily used within specific definition tables (see chapters 5 and following) describing certain specialized data model definitions. In case of elements, the presence indications “M”, “O”, and “C” are always meant relative to the respective parent element. I.e. if a parent element is optional (“O”) and a child is mandatory (“M”) the child element can only be present if the parent element is present as well.

The presence indications from the data model definitions describe general requirements on the presence of elements. These requirements can be strengthened (but not weakened) by process rules (these can be defined per so-called "featureType"). Please note that the indications and the aforementioned rules apply for "full messages" (so-called "full function exchange"). In contrast, the so-called "restricted function exchange" is designed to permit exchange of specific excerpts of data, i.e. fewer elements as potentially available from the data owner (partially even not all "mandatory" elements). This is discussed in more detail in section 5.3.4.

To give an example: We assume a data definition (“message”) “T” with an element “e” that is marked with “O” in the definition table of “T”. This means the message definition itself does neither require nor prohibit the presence of “e” in general. Furthermore, we assume a process “P” with rules for two different situations “Px” and “Py” is defined as follows: For “Px” the element “e” SHALL be set, whereas for “Py” it SHALL NOT be set. This means “T” is strengthened in the way that “e” SHALL be used in case of “Px” whereas it SHALL NOT be used in case of “Py”.

A counterexample shall explain that weakening the general requirements is not permitted: We assume the message “M” contains an element “d” which is marked with “M”. This means the data model definition REQUIRES the presence of “d”. In this case, no process can define any rule that permits the absence of “d” in an "full function exchange".

2.1.2 Specialized cardinalities

Many times, the possibility of a list is required. This means a specific element or data structure may sometimes occur more than one time in a contiguous sequence. For such parts, the proper list definitions in the XSD use the attributes

```
minOccurs="0" maxOccurs="unbounded"
```

This corresponds to the cardinality “0..unbounded” (or equivalently “0..infinity”). A given feature type may then restrict the upper or lower bound further. These restrictions are usually also shown in the tables describing certain specialized data model definitions. E.g. in the XSD a data structure “foo” may have the cardinality “0..unbounded” and a feature type “Bar” may define the use of “foo”, but with cardinality “1..20”.

2.1.3 Process dependent rules

Some feature types may define “process steps” or “circumstances” where further restrictions on the data model apply.

2.2 Common data types

The majority of data types use data types defined by W3C. In this specification these data types are identified by the namespace prefix “xs:” (“xs:boolean”, e.g.).

This specification defines also some own data types which are based upon W3C data types. In this specification these data types are mentioned without namespace prefix. Details on their definition can be found in section “Common data types” of the document [ResourceSpecification].

3 Architecture requirements

3.1 General rules

In theory, the SPINE data model permits arbitrary combinations of SPINE entities, SPINE features, SPINE classes etc. However, the definition of interoperable binding (see section 7.3) and subscription (see section 7.4) mechanisms requires imposing some restrictions, i.e. the definition of an architecture. Thus, the following rules apply:

1. A device consists of entities. An entity consists of (sub-)entities or features. For each address level (device level, entity level, feature level), the SPINE protocol defines an addressing scheme.
   Remark: This is explained in more detail in section 3.2.
2. Each device SHALL implement a so-called primary NodeManagement instance with information on itself. The device SHALL offer this information on its entity 0 (entity of the device with the entity address = 0) at feature 0 (feature of the entity 0 with the feature address = 0). This information SHOULD contain information about all entities of the device and all features of these entities. The so-called "entityType" (see section 7.1) of entity 0 SHALL be "DeviceInformation" (see also [ResourceSpecification], section "Entity Types"). The so-called "featureType" (see section 7.1) of feature 0 of entity 0 SHALL be "NodeManagement" (see also [ResourceSpecification], section "Feature Types").
3. This version of the specification does not consider any non-primary NodeManagement instance (i.e. NodeManagement instances on different addresses than entity 0 at feature 0 are not considered).
4. Each NodeManagement instance is a feature in general (see above). Thus, NodeManagement instances include proper information on itself or other NodeManagement instances as well, according to the rules given above.
5. On each feature there SHALL be at maximum one class implemented with regards to the features primary functionality. I.e. it is NOT permissive to offer more than one class on a single feature (see also section 5.3).
6. A feature on a device is assigned a "functional role". This role is EITHER "server" OR "client" OR "special". Please note a "functional role" is independent from any "connection role" (i.e. a role typically used in communications technologies like TCP).
7. The functional role "server" is used if the device is the "owner" of data of the corresponding feature (see also section 5.3.3). I.e. the device can notify changes or send this data as reply upon request. It may also accept "write" operations to perform a data change. In most cases, the feature role as "server" also includes the capability of a device to operate autonomously, i.e. to execute its server feature tasks even if no feature with role "client" sends data to control the server's feature tasks.
8. The functional role "client" is used to receive information provided by a "server" and to use the server's features accordingly (example: configure the server features using "write" operations, provided this is offered by the server feature).
9. The functional role "special" is reserved for specific features. It SHALL ONLY be used for features explicitly mentioned in official SPINE specifications. In this version of the specification the primary NodeManagement instance SHALL have the role "special".
10. The concepts for binding and subscription respect the role assignment. Details are given in the corresponding sections.

Note: The rules above only describe the architecture. Whether all or just reduced information of a device’s NodeManagement instances is shared with another device and whether the communication between the features of two devices is restricted (or even blocked) or not is discussed separately (esp. using the “trust level” concept).

3.2 Address level details

The concept of the miscellaneous address levels as described in section 3.1 by item 1 in section 3 is defined in more detail now.

The following example defines a device with name “someDevice” with three top-level entities (entities directly below the device “someDevice”) with entity addresses 0, 1, 4 (remark: “entity 0” describes a SPINE entity with the entity address = 0, feature 0 describes a SPINE feature with the feature address = 0):

```
“someDevice”
      +--- entity 0          (child of “someDevice”)
      |        +--- feature 0
      +--- entity 1
      |        +--- entity 4 (child of “someDevice”/entity 1)
      |        |        +--- feature 7 (*1)
      |        +--- entity 5 (child of “someDevice”/entity 1)
      |        |        +--- feature 1 (*2)
      |        |        +--- feature 7 (*2)
      |        +--- feature 1 (child of “someDevice”/entity 1)
      |        +--- feature 4 (child of “someDevice”/entity 1)
      +--- entity 4           (child of “someDevice”)
      |        +--- feature 1 (child of “someDevice”/entity 4)

(*1): child of “someDevice”/entity 1/entity 4
(*2): child of “someDevice”/entity 1/entity 5
```

In this example, the top-level entity 1 contains two features (1, 4), but also two other (sub-)entities. This shows that entities can be nested.

Starting at the top, the full address paths are (note that this is an example to describe the SPINE address level concept and that the following lines DO NOT state valid SPINE addresses; the correct usage of SPINE addresses is described in chapter 5):

1. device “someDevice”
2. device “someDevice” / entity 0
3. device “someDevice” / entity 0 / feature 0
4. device “someDevice” / entity 1
5. device “someDevice” / entity 1 / entity 4
6. device “someDevice” / entity 1 / entity 4 / feature 7
7. device “someDevice” / entity 1 / entity 5
8. device “someDevice” / entity 1 / entity 5 / feature 1
9. device “someDevice” / entity 1 / entity 5 / feature 7
10. device “someDevice” / entity 1 / feature 1
11. device “someDevice” / entity 1 / feature 4
12. device “someDevice” / entity 4

13. device “someDevice” / entity 4 / feature 1

Only the full address path is unique. I.e. feature 7 of { device “someDevice” / entity 1 / entity 4 } differs to feature 7 of { device “someDevice” / entity 1 / entity 5 }. Similarly, entity 4 of { device “someDevice” / entity 1 } differs to entity 4 of { device “someDevice” }.

It shall now be explained how this is modelled in the SPINE XSDs and XMLs. In the XSDs an element “entity” is often defined with attributes

```
minOccurs="0" maxOccurs="unbounded"
```

The upper bound of this cardinality means an “entity” tag can occur arbitrarily often at the given position in an XML (representing arbitrary depth of nested entities). However, a device that complies with this and previous versions of the SPINE specification only can silently discard messages where an entity list comprises more than 15 "entity" items. This means an implementation SHOULD avoid to implement any of its entities with more than 15 "entity" items as it is likely that a communication partner will ignore such entities. The lower bound “0” of this cardinality means there may be no “entity” element at all. Please note that the architecture itself always requires the presence of at least one entity. But certain message definitions that contain address elements with this generic address type may well permit the absence of “entity” elements for specific purposes (among others, the deletion of all so-called “bindings” between two devices omits all “entity” elements in a specific address element).

The i-th occurrence of “entity” within an address instance corresponds to the i-th entity level (depth). The entity level “i+1” is a child of entity level “i”. I.e. an XML part with full address path for { device “someDevice” / entity 1 / entity 4 / feature 7 } is represented as follows:

```
<device>someDevice</device>
<entity>1</entity>
<entity>4</entity>
<feature>7</feature>
```

Within the message description tables, the possibility of multiple “entity” tags (representing nested entities) is specified as follows:

... entity (list)

As already explained above, multiple entity tags in an address instance always describe “nested entities”, i.e. entities as child elements of other entities. Such multiple entity tags DO NOT describe “parallel” entities (entities on the same hierarchical level). One address instance can only describe one single address path and is not used to describe several parallel address paths of a device.

4 Compatibility considerations

4.1 Introduction

SPINE data models are based on XSDs (XML Schema Definition) files as well as on additional specification documents (this document and [ResourceSpecification]).

This chapter focuses on compatibility aspects related to the use of different versions of the SPINE data models defined by the SPINE XSD files. Details on the versioning can be found in Annex C.

Achieving and preserving compatibility of data among different versions belongs to the most underestimated topics of data modelling and software development. For proprietary (i.e. “closed”) developments this is certainly an issue as well, however, due to the proprietary nature of the product/definition solutions can typically (but not always) be achieved rather easily if there are options to apply “ugly workarounds”. For standardized or public protocols or definitions the situation becomes far more difficult. Considering compatibility requirements and mechanisms from the beginning on helps maintaining both data modelling (improvement and further development of a data model) as well as product development with a minimum risk of breaking data processing from distinct versions.

Suppose an XML document was created based upon the SPINE data model of version “X”. Questions arise whether or how this XML document can be processed on a system supporting SPINE data models of version “Y”. Such considerations lead to two primary aspects:

1. Requirements on XML processors:
   What must an XML processor do or consider in order to process an XML document of an “old” or “unknown” version.
2. Requirements on the SPINE data model development:
   What must be considered by data model (XML schema) developers in order to achieve and preserve compatibility among different versions of the schema. This has a direct impact on XML processor requirements.

4.2 Notations and definitions

Definition of the term “ordered”: Two distinct numbers a, b are ordered if exactly one of the following relations applies:

- a < b
- a > b

Definition: Schema version numbers are assigned with a “natural order”. This means the successor of an already present schema SHALL always be assigned a greater version number than the version number of its preceding schema.

Definition: Let v_a and v_b be ordered numbers denoting the version of the SPINE schema (the SPINE data models). Furthermore, we define v_a < v_b, i.e. v_b represents a “newer” schema version than v_a.

Remark on the term “newer”: A successor is always newer than its adjacent predecessor. If any two schema versions v_a, v_b with v_a < v_b are considered it cannot be said in general that v_b is “newer” if the time of its creation is meant. However, to simplify explanations we suppose a “linear (chronological) development” of the schema versions in time.

Definition of the term “valid”: An XML document is considered valid against a specific schema if well-defined rules for the validation are fulfilled. (These rules are defined separately.)

Remark on the term “valid”: A new version may define additional content, leading to XML documents that are unknown completely or partly with regards to an old version. Validity does NOT mean that an XML document can be evaluated (“understood”) completely. It just means that “known” parts can be evaluated according to the validity rules.

Definition: A SPINE schema of a specific version comprises of all SPINE data models and documents belonging to the version. This means a SPINE schema revision is always “complete”. As a consequence, it is NOT considered to process XML documents comprising of parts stemming from different SPINE schema versions. I.e. “mixing” SPINE schema versions is NOT valid.

Definition of the term “backward compatible”: A schema version v_b is backward compatible if any XML instance of schema version v_a remains valid against schema version v_b.

Definition of the term “forward compatible”: A schema version v_a is forward compatible if any XML instance of schema version v_b remains valid against schema version v_a.

Remark: Forward compatibility is more difficult to achieve in general.

On the use of the term “compatible”: Unless stated otherwise, the term “compatible” is used to express that two schema versions are both forward compatible AND backward compatible.

Definition of the term “cardinality change”: Within XML schema an element “E” is associated with an explicit or implicit property “maxOccurs” with proper value. The term “cardinality change” denotes the case where macOccurs of “E” is at least 1 for two schema versions v_a and v_b, but different between v_a and v_b.

4.3 Compatibility Rules

4.3.1 Introduction

As already mentioned in section 4.1 data model compatibility is a complex subject. In order to cope with the complexity a number of “basic compatibility rules” for the further development of the SPINE data model (especially SPINE XSDs) are defined.

In addition to basic compatibility rules some extensibility mechanisms of XML schema are discussed.

Please note that this subject in general covers aspects that are relevant for the specification development itself (i.e. the further development of the SPINE specification) as well as for implementers.

4.3.2 Brief comment on compatibility issues with XML schema and implementations

The SPINE data models are formulated in XML schema. When this document was created two versions of XML schema were available: XML schema 1.0 and XML schema 1.1. The majority of applications make use of XML schema 1.0 as XML schema 1.1 is rather new.

XML schema 1.0 already provides some features permitting additional content (additional elements) in an XML, i.e. content that is not explicitly defined in a given XML schema. However, these so-called wildcards have not been defined in a way that compatibility of schemas can be formulated across multiple versions sufficiently. In fact, versioning has not been covered properly in XML schema 1.0 and there are plenty different guidelines/workarounds and opinions how to deal with this situation.

With XML schema 1.1 this situation improved a lot as wildcards have been modified accordingly and a new feature named “open content” was defined. Anyhow, a deeper analysis reveals that certain aspects of compatibility are still an issue and cannot be formulated easily using XML schema solely.

This is not necessarily a lack of XML schema features. Some problems reported on miscellaneous blogs can be classified as unawareness of compatibility requirements that did result in implementations not designed to cope with different versions. Other problems can best be classified as unavailability of tools or libraries designed for compatible schema versions. Of course, these kinds of problems can hardly be solved just by specification and are beyond the scope of chapter 4.

As an outcome of the analysis, the subsequent sections (and esp. section 4.3.4) present rules and guidelines which make use of different approaches.

4.3.3 Basic rules

4.3.3.1 EEBus Initiative e.V. as SPINE specification authority

Only data models owned and originated by the EEBus Initiative e.V. AND defined for the “SPINE interface” can be called “SPINE data models”.

Remark: The EEBus Initiative e.V. can develop further definitions (e.g. “SHIP”). Definitions from these developments are NOT considered SPINE data models. If definitions from these developments shall become SPINE data models they need to be brought into the SPINE data model development and release process in order to become applicable for the SPINE interface.

Only the EEBus Initiative e.V. is permitted to release SPINE schemas.

4.3.3.2 Modifications

A SPINE schema must not be modified in any way except for modifications permitted in chapter 4 explicitly.

4.3.3.3 SPINE namespaces

The EEBus Initiative e.V. is the owner of one or more so-called namespaces reserved for SPINE data models. For each SPINE schema version one of the SPINE data model namespaces is chosen as so-called targetNamespace. The SPINE data models of a SPINE schema version are assigned to this targetNamespace.

Only the EEBus Initiative e.V. is permitted to assign to SPINE data models a SPINE namespace.

An XML document that shall be considered valid against a SPINE schema must use namespaces offered by the SPINE schema only. Additionally, it must only contain content as specified by the SPINE data model of a given version.

4.3.3.4 Use of other namespaces or schemas

The SPINE data models are based upon a limited and well-defined set of schemas: This is primarily the W3C XML schema definition. In order to provide certain definitions for compatibility/validation purposes a W3C versioning schema can be used as described later on. The SPINE schema does not use or import any other schema.

4.3.3.5 Releases and branches

The EEBus Initiative e.V. can provide two kinds of releases of a SPINE schema: Official releases (versions) and unofficial releases. Unofficial releases are used during the development of a SPINE schema towards an official release. Unofficial releases are NOT constrained by compatibility requirements, are NOT supported with regards to compatibility, and are NOT considered legal in any product or solution. Throughout this document only official releases are considered, unless stated otherwise.

The EEBus Initiative e.V. provides exactly one branch with official SPINE schema releases. All versions of this branch are ordered as described in section 4.2. These versions shall be developed to provide compatibility against each other.

Remark on the aforementioned “one branch”: This basically means the SPINE data models are developed in a “linear way”, i.e. there is no branch-off and no “parallel” development or variant of a SPINE schema.

In theory it can happen a new version (denoted here as Vx) needs to break compatibility to all previous versions. Of course, this should be avoided. But if this happens successors to Vx shall be developed to be compatible to Vx. See section 4.3.4.2 for details.

4.3.3.6 Further aspects

Aspects on so-called “data binders” are not considered throughout chapter 4.

XML is just an example of a data instance matching a schema. Rules described in chapter 4 apply for equivalent content types as defined by the EEBus Initiative e.V. as well (e.g. JSON could be a candidate for a compatible type).

4.3.4 Technical rules

4.3.4.1 Basic concept

This section shall just give a brief idea on the SPINE version compatibility concept.

Suppose version 1 permits this XML:

```
<person>
      <name>
            <firstName>your first name</firstName>
            <lastName>your last name</lastName>
            <nickName>your nick name</nickName>
      </name>
</person>
```

A very common method permits adding new elements at the end of already known elements of version 1. This means new elements can be defined beyond the last element of a structure. Thus, the schema version 2 could be extended with elements “title” and “address” to permit this XML:

```
<person>
      <name>
            <firstName>your first name</firstName>
            <lastName>your last name</lastName>
            <nickName>your nick name</nickName>
            <title>your title</title>
      </name>
      <address>...</address>
</person>
```

Similarly, version 3 could define an additional element beyond “address”, e.g.

Skipping unexpected elements beyond expected elements is a rather simple task to create compatible content. It should be noted that extensions at any place of the XML schema (more precisely: before the last possible well-defined element of a structure) are NOT supported for SPINE standard classes (see below for SPINE complex classes)! Among others, this helps definition and maintenance of compatible non-XML content models in the future (this can esp. become relevant for binary formats defined with ASN.1, e.g.).

For complex classes another aspect needs to be considered, leading to a different rule. The function of a complex class basically permits larger/deeper structures, consisting of (usually renamed) functions of non-complex classes. In many cases the types of the non-complex class functions are used with restriction in order to reduce the number of elements of interest. As brief example we consider a complex class function “friends” where the non-complex function “person” is reused as list, but with the restriction that “lastName” is discarded. A proper XML could look like this:

```
<friends>
      <person>
            <name>
                  <firstName>Fred</firstName>
                  <nickName>Boss</nickName>
            </name>
      </person>
      <person>
            <name>
                  <firstName>Anna</firstName>
            </name>
```



```
	</person>
</friends>
```

In a subsequent version it might be required to have “lastName” again. This requires reducing the restriction that was applied with the previous version as it is not feasible to append another “lastName” in the type definition of the non-complex class “person”. Thus, the subsequent version permits the following XML:

```
<friends>
	<person>
		<name>
			<firstName>Fred</firstName>
			<lastName>Smith</lastName>
			<nickName>Boss</nickName>
		</name>
	</person>
	<person>
		<name>
			<firstName>Anna</firstName>
		</name>
	</person>
</friends>
```

Here, a “new” element appears “in between” and not just at the end. However, such kind of extensions should only occur for the case described above (i.e. reduce a restriction in a subsequent version).

Apart from details on the kind of element extensions further aspects need to be considered in detail: Extensions of enumerations, relations of namespaces and versions, processing vs. validation scopes, etc. These topics are discussed in the subsequent sections.

4.3.4.2 Version compatibility groups

A version compatibility group is defined as a closed interval [v_min_i, v_max_i] where “i” is the index of the interval and v_min_i and v_max_i are version numbers with v_min_i ≤ v_max_i. Furthermore, schemas of this group SHALL be compatible.

The intervals of two groups SHALL NOT overlap! I.e. they SHALL NOT have any version number in common.

4.3.4.3 Namespace format, versioning and location

Some versioning concepts change the namespace with each version. This, however, leads to a broken compatibility in general. Consequently, it is most recommended to NOT change a namespace with every schema version. This concept is used for SPINE as well.

For official SPINE versions a version number format “major.minor.revision” (2.7.3, e.g.) is used. For each version compatibility group “i” the lower interval boundary v_min_i shall be used in a simplified format to denote the basic namespace of the interval group.

The first official release will be “1.0.0”, simplified as “1”. The namespace of the corresponding first group is defined as http://docs.eebus.org/spine/xsd/v1

Remark: Unofficial versions have a different namespace format.

Supposed there are two version compatibility groups with the intervals [1.0.0, 2.7.4] and [3.0.0, 3.5.1]. The second group is then assigned the namespace

http://docs.eebus.org/spine/xsd/v3

All schemas of a version group SHALL use the group’s assigned namespace as targetNamespace. XML documents that shall be compatible throughout a group SHALL use this namespace as well.

If compatibility is not required the schema’s (full) version can be used as namespace. Example: For version 2.7.4 this would be “http://docs.eebus.org/spine/xsd/v2.7.4”. Note that such namespaces are NOT supported to achieve compatibility (or interoperability between devices/applications). This means they can only be used for “internal purposes”.

4.3.4.4 Version identification in schema files

As a consequence of section 4.3.4.3, throughout a compatibility group neither the schema nor suitable XML documents can denote the “real” version by the namespace. In practice knowledge of the real version is often required (e.g. for pre-processing in order to skip unknown elements of the XML before the version specific XML processor is executed). Of course, schemas of different versions must be distinguishable as well.

For XML documents the real version of the corresponding schema can be expressed through the element “datagram.header.specificationVersion” (see section 5.2.7). Additionally, the SPINE class “Version” can be used to express a SPINE specification version. This is not further detailed in chapter 4. The attribute “xsi:schemaLocation” is also not further discussed for versioning.

The full version of a SPINE schema is expressed using the attribute “version” of the element “schema”. For a schema version “2.3.52” (associated to compatibility group “1”) a basic preamble could look like this:

```
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
      xmlns:ns_p="http://docs.eebus.org/spine/xsd/v1"
      targetNamespace="http://docs.eebus.org/spine/xsd/v1" version="2.3.52"
      elementFormDefault="qualified">
```

Please note subsequent sections will show further attributes of the SPINE schema preambles.

4.3.4.5 Schema files and location

Many schema concepts define the namespace format in URI-style. For some of these concepts the schema files are also offered for download at the given URI. This is not offered for the SPINE schema files right now.

For practical reasons schema files of a given version are typically stored at a certain location (a local folder, e.g.). The current concept does NOT put a version number in the schema file names. Many

SPINE schema files contain one or more “include” instruction to other SPINE schema files of the same version. These “include” instruction use relative paths.

As a consequence, for each version all schema files are expected to be stored at an individual (i.e. version dependent) location.

4.3.4.6 defaultOpenContent

Among others, for a formal description of extensibility and processing rules the feature “defaultOpenContent” is used in this document. This feature was introduced with XML schema 1.1. A defaultOpenContent element must be placed after “include” instructions and before content definitions (types, elements). Different kinds of defaultOpenContent instances are used within this document and referred as DOC1 and DOC2, resp.:

DOC1:

```
<xs:defaultOpenContent mode="suffix">
      <xs:any namespace="##targetNamespace" processContents="skip"/>
</xs:defaultOpenContent>
```

DOC2:

```
<xs:defaultOpenContent mode="suffix">
      <xs:any namespace="##targetNamespace" processContents="skip"
      notQName="##definedSibling"/>
</xs:defaultOpenContent>
```

The subsequent sections explain when and how these definitions apply.

Remark on processing XML schema 1.1:

As defaultOpenContent is unknown to XML schema 1.0 some schema processors either cannot support it at all or need to be informed explicitly to process an XML schema 1.1. Some of these tools can be configured to unconditionally process an XML schema with XML schema mode 1.1. Some of these tools can switch to XML schema 1.1 processing from a proper announcement in the XML schema preamble. To give an example, such an announcement can consist of these definitions:

```
vc:minVersion="1.1" xmlns:vc="http://www.w3.org/2007/XMLSchema-versioning"
```

4.3.4.7 Formal description of element extension

The SPINE schema files still make use of XML schema 1.0. As already mentioned in section 4.3.2, XML schema version 1.0 does not sufficiently support formulation of extensible schemes. This means SPINE schema files of a specific version represent only the explicitly defined structures and elements. I.e. from these files only it cannot be deduced which kind of additional content is permitted and how it shall be processed.

Throughout this section we assume the following: An application is designed for a single schema version “v_app” but shall be compatible within its compatibility group. The version “v_app” belongs to compatibility group “i”, hence v_min_i ≤ v_app ≤ v_max_i. The application processes an XML of schema version “v_xml” which belongs to the same compatibility group.

For a validation of an XML a modified set of schema files is required: The “modified schema files” are identical to the original schema files with the following exception: In complex classes, types of non-complex class functions are used without any additional restriction (the background for this rule is explained in section 4.3.4.1).

An application SHALL process the XML as if all modified schema files of version “v_app” contain defaultOpenContent variant DOC1.

Remark: A brief (but not accurate) explanation shall help understanding the meaning of DOC1: Among others, this means an application must ignore unknown elements that appear beyond the last explicitly specified element of a structure. In addition, DOC1 imposes rules on the namespace of the unknown element. It also clearly states that occurrence of unknown elements at other positions (i.e. before or between explicitly specified “adjacent” elements) classifies the XML as invalid.

Afterwards, an application should ignore those unexpected element instances that arose from a cardinality change (see section 4.2 for the definition).

Remark: This means an application is not forced to accept content with list sizes greater than the maximum list sizes specified in v_app.

The subsequent pictures show some examples. Please note these examples are not exhaustive. For each example the definition of a given schema version is shown. Additionally, it is drawn which kind of extension could occur (or cannot occur) with a succeeding schema version.

The green boxes show potential extensions with a succeeding schema version (within “in” beyond “max”; within “out” beyond “max”; beyond “out”):

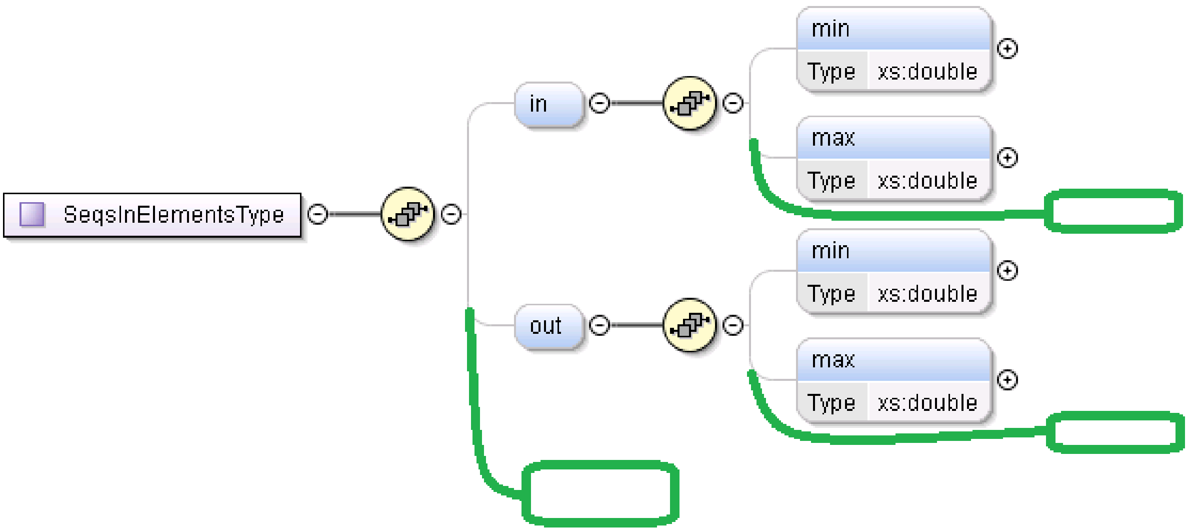


Figure 1: Element extension example 1

The next example shows potential extensions in case of derived types as also used within some SPINE definitions (beyond “b”; beyond “xyz”). Please note this means an XML of a subsequent schema version may contain a tag between the tags “b” and “specificData”.

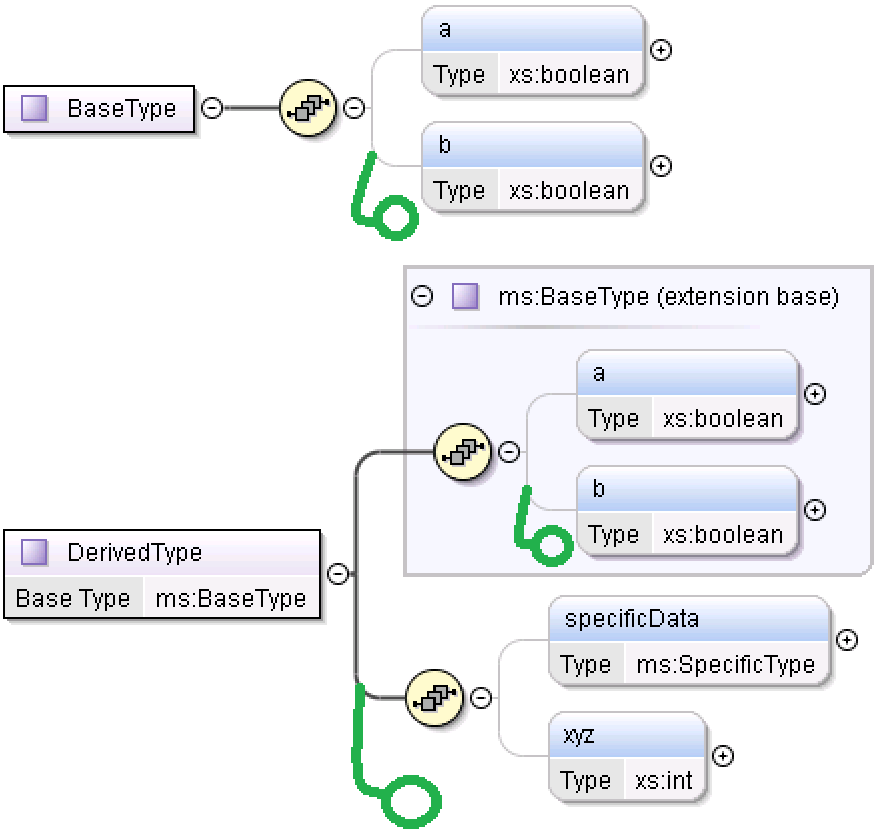


Figure 2: Element extension example 2

The next example shows where a subsequent schema version must not introduce new elements in non-complex class functions (before “in”; within “in” between “min” and “max”; between “in” and “out”; within “out” before “min”):

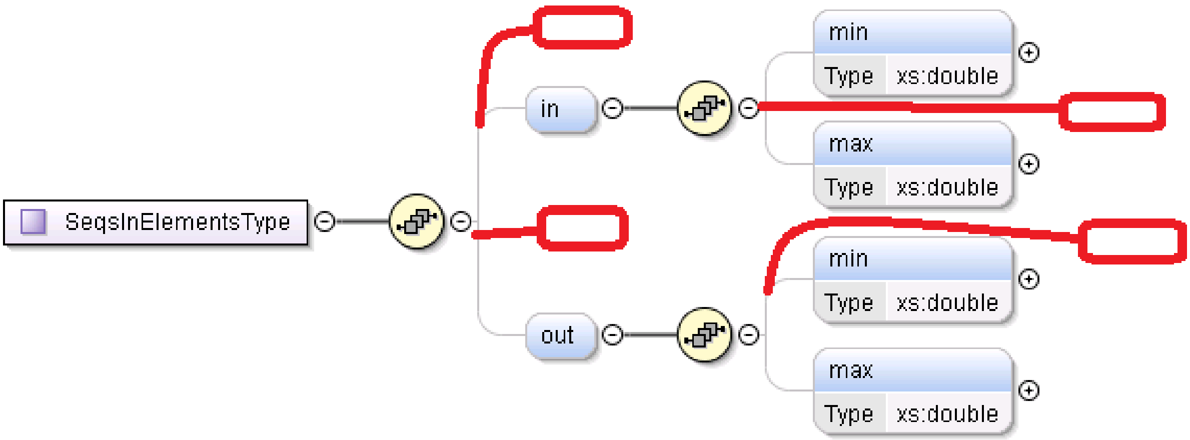


Figure 3: Element extension example 3

Please remind the examples are not exhaustive.

4.3.4.8 Value set extensions

With XML schema enumerations as well as other kinds of value sets can be imposed on basic types. Within this section we consider a type T. The set of all possible values of type T is denoted as S. For a schema version v_a the set is S_a and for version v_b it is S_b. Furthermore, we assume v_a < v_b.

Changing a value set from one version to another is always problematic. If S_a contains values unknown to version v_b backward compatibility is broken. If S_b contains values unknown to version v_a forward compatibility is broken.

This situation is comparable to the case of removed or added elements. For elements some flexibility with regards to compatibility can be expressed by XML schema language using defaultOpenContent. For value sets this is not as easy (or at least not practical).

Thus, compatibility rules are defined as follows:

An application SHALL be prepared to encounter elements with values not permitted by the application's SPINE schema version.

An application can discard elements with unknown values. In case the discarded element is essential (for reasons not specified in this document) the application can discard up to the whole XML. It can treat the XML as invalid.

4.3.4.9 Rules for schema development

In order to maintain compatibility across as many versions as possible some rules need to be considered during schema development. As this is a complex topic just a few of them are explained here briefly.

Basically, reordering and renaming of elements is prohibited.

The value range of a type must not be reduced with a newer schema version. But even the extension of a value range is problematic as previous schema versions and applications will not be able to support new values.

With regards to non-complex classes a new schema version shall be defined as if all previous schema versions and the new schema versions are defined with defaultOpenContent as specified by DOC2. This also means compatibility aspects must be considered and evaluated as if DOC2 was present since the first version. With regards to complex classes these rules basically apply as well, but they are relaxed as follows: In a newer version there may be less restrictions in the derivation of the non-complex class types.

Care must be taken if anonymous sequence or choice definitions are used! Without uniquely embracing type name (and usually also element name, if the element is of this type), this can lead to definitions where a choice/sequence cannot be extended with a subsequent schema version.

For complex type definitions with root compositor "xs:choice" it is recommended to configure the choice with the attribute

```
minOccurs="0"
```


In case of a definition of “xs:group” with root compositor “xs:choice” there is no need to make the choice optional as the group itself can be referenced with attribute minOccurs=“0”.

Further use of anonymous sequence or choice definitions need to consider and explain compatibility rules including implementation guidelines for applications explicitly. Especially the definition of “payload” requires special attention.

As explained in section 4.3.4.8 modifications to value sets are problematic. SPINE classes or SPINE feature type definitions should consider this case and define explicitly how to deal with unknown values on a detailed level.

4.3.4.10 Brief comment on validation

The previous sections explain difficulties with XML schema to formulate compatible schemas. Several rules are defined how to deal with unexpected content. As a consequence, a strict validation of an XML against the official SPINE schema of a specific version of the same compatibility group can only be performed after all unknown elements and elements/content with unknown values have been removed as described.

4.3.5 Further aspects

Some non-SPINE concepts require applications with multiple versions. Other concepts define the exchange of a schema between two endpoints. Such concepts are not considered further in chapter 4.

The definition of a version negotiation is a very common part of protocols and helps or even permits establishment of compatible communication between two participants. The definition of a version negotiation is beyond the scope of chapter 4 (see section 7.1 for details). But it shall be noted that such a negotiation typically reduces the problem of value set changes (see 4.3.4.8) as afterwards the participants would only use values supported by both of them.

4.4 Conclusion

Future versions of the SPINE specification may define further immediate child elements for “payload”. This can result in the definition of a new sub-group of “PayloadContributionGroup” or in an extension of any of its already described sub-groups. Within an XML it is not possible to detect to which group an extension belongs. This must be considered for each kind of extension of “payload”. In general, the following rules apply:

1. A parser of device “A” complying with this SPINE version shall skip any immediate child of “payload” that is not defined by this SPINE version (i.e. just this unspecified element shall be skipped, not the payload segment it belongs to).
2. A device “B” complying with a SPINE version that defines a new immediate child of “payload” shall interact with a device “A” in a compatible way. I.e. device “B” shall not expect device “A” to evaluate this new child.

5 SPINE Datagram

5.1 Introduction

5.1.1 General information

The ISO-OSI layer model defines an application layer, among others. Interaction with another application is considered an end-to-end connection. With regards to the SPINE data model the "functions" could be exchanged on this level.

In order to model the exchange between two end points dynamically the SPINE data model defines an element "datagram" consisting of "header" and "payload".

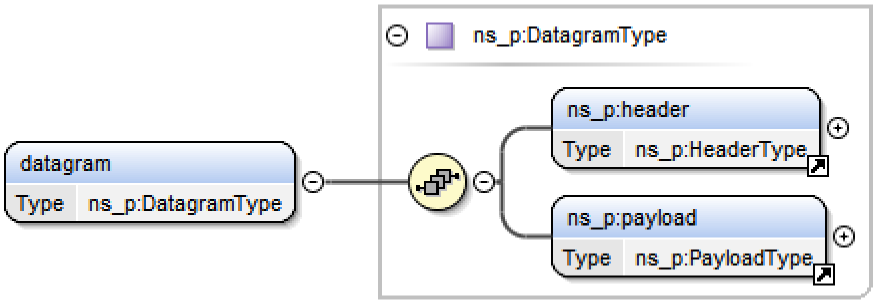


Figure 4: SPINE datagram

5.1.2 Structure

The structure of the SPINE datagram SHALL be set as shown below.

Element name	M/O/NV/C (see 2.1.1)	Brief explanation
datagram	M	The root element of the SPINE datagram SHALL always be present.
datagram. header	M	The header element SHALL be present. For sub-elements of the header see section 5.2.7.
datagram. payload	M	The payload element SHALL be present. For sub-elements of the payload see section 5.3.2.

Table 1: Structure of the SPINE datagram

The notation "a.b" (for example "datagram.header") is used here to show the hierarchical structure of a SPINE Datagram. It is used to define that "b" is a child element of "a".

5.2 Header

5.2.1 General information

The SPINE header contains information on the version of the applied data model, addresses of end points, etc.

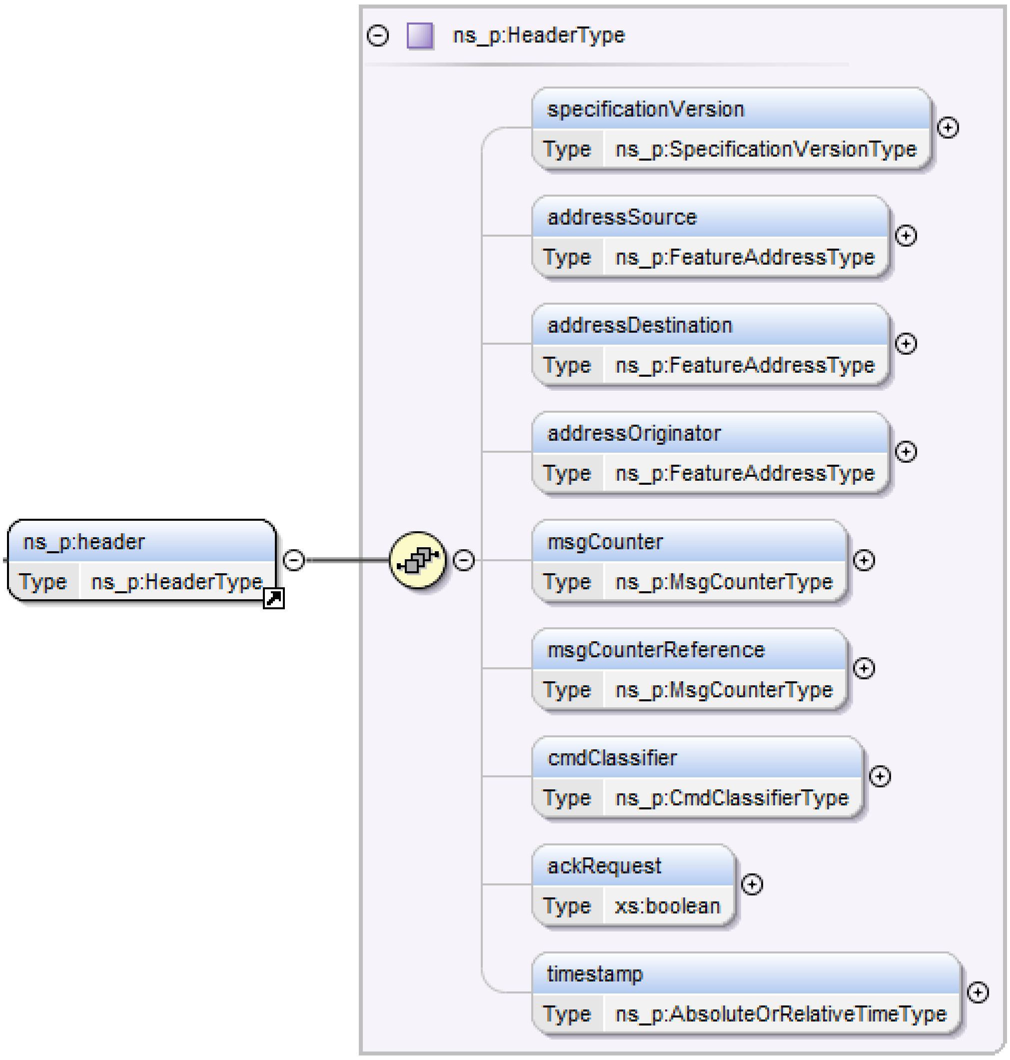


Figure 5: SPINE header

5.2.2 Address information

5.2.2.1 addressSource and addressDestination

The elements addressSource and addressDestination are defined with their child elements and purpose in Table 3. addressSource corresponds to the unique address path (see section 3.2) of a feature where a SPINE message was created. addressDestination corresponds to the unique address path of the receiving feature. Subsequently some rules on the child elements of addressSource and addressDestination are given.

If a device creates a message it SHALL set its entity and feature address parts into addressSource. The recipient's entity and feature address parts SHALL be set in addressDestination.

The “device” address parts of addressSource and addressDestination may be omitted in some cases. When to set the “device” address parts is defined subsequently. For this, we consider a communication between the SPINE devices “A” and “B”. The “common rules” (see below) apply in any case. Furthermore, additional rules apply dependent on the communication mode:

Common rules on the use of "device" address elements:

In general, the use of "addressSource. device" and "addressDestination. device" REQUIRES to ensure the uniqueness of all "device" values. I.e. device "A"'s value of "device" SHALL be different to device "B"'s value of "device". This is also in accordance with section 7.1.1.5.1.

We assume a message is transmitted from device "A" to device "B":

1. If "addressSource. device" is present and not empty it SHALL be identical to device "A"'s own value of "device". The message SHALL be considered illegal if the value differs.
2. If "addressDestination. device" is present and not empty it SHALL be identical to device "B"'s own value of "device". The message SHALL be considered illegal if the value differs.

Additional rules in case of "simple communication mode":

The following rules apply in case of a "simple communication mode" (see section 6.1). In this mode, two devices are considered being directly connected to each other. We assume a message is transmitted from device "A" to device "B":

Within the header the "device" elements of addressSource and addressDestination SHOULD be set to the correct value. The absence of "device" in "addressSource" SHALL be treated as if "device" in "addressSource" was set to device "A"'s own value of "device" (i.e. the "device" address part of device "A"). The absence of "device" in "addressDestination" SHALL be treated as if "device" in "addressDestination" was set to device "B"'s own value of "device" (i.e. the "device" address part of device "B").

Additional rules in case of "enhanced communication mode":

The following rules apply in case of an "enhanced communication mode" (see section 6.2). In this mode, both "addressSource. device" and "addressDestination. device" SHALL always be set in the header.

5.2.2.2 addressOriginator

As explained in Table 3 the element "addressOriginator" can be used to model information of a "forwarded" XML and permits preserving the address of the original submitter. The use of addressOriginator is optional. However, a brief example shall demonstrate a typical use of this element:

We assume device "A" created a message "X" and submitted it to device "B". We also assume device "B" serves as some kind of "data warehouse" for all messages it received from miscellaneous devices. With its "data warehouse" service it can provide an overview of messages from all its connected devices. We also assume these messages are available on one or more features of device "B". If a device "C" reads on such a feature of device "B", device "B" must set its own feature address into "addressSource" of its response to device "C". In order to keep the information where the functional content of the message originally stemmed from (device “A”) the element “addressOriginator” needs to be set properly.

Note: If "addressOriginator" is used as described above, the elements "addressSource. device", "addressDestination. device" and "addressOriginator. device" SHALL be set!

5.2.3 Message counter

A message counter (msgCounter, msgCounterReference) serves for the identification of a message. This is especially important if replies are delivered in a different order than the corresponding requests.

5.2.3.1 msgCounter

If a device creates a message it SHALL assign msgCounter a value that does not conflict with any other of its recently created messages (i.e. it SHALL be a virtually unique value). In general, the value SHALL NOT collide with any of the device’s previously created messages where the device still expects proper responses. The msgCounter value SHALL be ascending and restart from 0 once the largest possible number (2^{64}-1) was used. The msgCounter values MAY skip some numbers (e.g. after a message "X" with msgCounter "10" a message "Y" with msgCounter "15" is sent).

If a SPINE device "A" receives a message "X" from SPINE device "B" with a msgCounter less or equal than the last msgCounter received from device "B", "A" SHALL process the message "X" as usual. Afterwards, device "A" SHALL use the unexpectedly low msgCounter value as the last msgCounter received from device "B". If device "A" receives a message with unexpectedly low msgCounter value from device "B", it MAY report this to the user in order to report that the communication partner may have a problem or unexpected condition (e.g. factory reset); however, such a report is only useful if the underlying communications technology preserves the order of messages!

Implementation advice: A best practice to keep msgCounter values unique even in case of power failures is as follows: A device keeps a "stored msgCounter value" in a non-volatile memory. When a device is turned on, it first increases the "stored msgCounter value" by 1000 and uses the result for its next message. From then on, every 1000th created message the device copies the msgCounter value into the "stored msgCounter value".

Please note: A device MAY assign its messages in the msgCounter field values that are unique across all its communication partners. But this kind of uniqueness is not required by this specification. Rather, this kind of uniqueness may be easier for implementation in embedded devices whereas web servers may scale better with values that are communication partner specific.

5.2.3.2 msgCounterReference

If the created message serves as reply or comparable reaction to a previously received message, the device SHALL take the value of msgCounter from the received message and set it into msgCounterReference of its own message.

The element msgCounterReference SHALL NOT be set if the sent-out message does not relate to a received message.

5.2.4 Message classifiers

The element cmdClassifier conveys information on the kind of operation associated with the given message “M”.

This section also uses the terms “dedicated reply”, “dedicated data”, or “dedicated response”. The SPINE specification defines a number of (sub-)classes with (SPINE) functions. A device “A” may be the “owner” of a specific function instance (i.e. it has the role “server” at the function’s feature address). Another device “B” may request to get a (full or specifically curtailed) copy of this (SPINE) function. A response is called “dedicated” if this request can be fulfilled, i.e. the reply conveys indeed a proper copy of this (SPINE) function.

Table 2 shows permitted values for cmdClassifier and the relation to the kind of message “M”. The receipt of a message “M” may lead to the return of a related “acknowledgement message” (see section 5.2.5), denoted as “N”. The scope of “N” is also shown in Table 2.

cmdClassifier of message “M”	**Kind of message “M”**	**Scope of related acknowledgement message “N” (see section 5.2.5.2)**
read	Initial	Application error
write	Initial	Application success or error
call	Initial	Application success or error
reply	Response	Transmission success or error
notify	Response	Transmission success or error
result	Acknowledgement	Not applicable

Table 2: cmdClassifier values and kind of messages for a message “M” and the scope of related acknowledgement messages

We assume device “A” sends an “initial message” to device “B” (address level details like “feature” are just left out to simplify the explanation). Dependent on the received “initial message” device “B” creates a “response message” for device “A”. I.e. a “response message” is always related to an “initial message”. Furthermore, “response messages” are only used in case the received “initial message” was processed successfully by the recipient.

An “acknowledgement message” can be used to indicate whether an “initial message” or “response message” was processed or transmitted successfully or not (see section 5.2.5.2).

The different values of cmdClassifier are used for the following operations:

read

This denotes a “read operation” from the sender (addressSource) to the recipient (addressDestination). The recipient is considered the owner of the proper information (i.e. function). The recipient SHALL respond with a dedicated "reply" if the read operation is valid and can be processed regularly, i.e. without any failure. Otherwise the recipient SHALL respond with an “acknowledgement message” (see section 5.2.5.2) where “errorNumber” is set to a different value than “0”. Please note that the obligation to create a “reply” or “error indication” is independent from the value of the element “ackRequest” of the header. More precisely, the acknowledgement request (see section 5.2.5.1) of a received “read operation” SHALL NOT be evaluated.

reply

This denotes a “reply operation” from the owner of the (assumed) proper information (addressSource) to the recipient (addressDestination). A reply SHALL be created according to the rules given for cmdClassifier value "read". It SHALL NOT be used in any other case. The recipient of a “reply operation” SHALL evaluate the acknowledgement request of the message according to section 5.2.5.1. Please note that a proper acknowledgement message just denotes a transmission success or error of the “reply operation”.

notify

This denotes a notification (“notify operation”) from the owner of the proper information (addressSource) to the recipient (addressDestination). In contrast to a dedicated “reply” upon a “read” message, a “notify” message is created autonomously by the owner of the proper information, i.e. without the need of a received “read” message. A typical example for the creation of a “notify” message is the notification of a value change in a feature that is subscribed by the recipient. The recipient of a “notify operation” SHALL evaluate the acknowledgement request of the message according to section 5.2.5.1. Please note that a proper acknowledgement message just denotes a transmission success or error of the “notify operation”.

write

This denotes a replacement or modification instruction (“write operation”, “full” or “restricted”) from the sender (addressSource) to the recipient (addressDestination). The recipient is considered the owner of the information (SPINE class function) that shall be replaced or modified according to the instruction. The recipient of a “write operation” SHALL evaluate the acknowledgement request of the message according to section 5.2.5.1.

call

This denotes an instruction (“call operation”) from the sender (addressSource) to the recipient (addressDestination) to trigger a specific action at the recipient. A “call operation” can be used to exchange information where the “ownership concept” of the other classifiers usually does not apply. The recipient of a “call operation” SHALL evaluate the acknowledgement request of the message according to section 5.2.5.1.

result

This message classifier is used for so-called "application acknowledgement messages" (also called "result messages"; see section 5.2.5.1) to indicate the general success or error with regards to a transmitted message. The recipient of a “result message” SHALL NOT evaluate the acknowledgement request of the received “result message”. A "result message" SHALL NOT be created as any kind of response to a received "result message".

5.2.5 Acknowledgement concept

5.2.5.1 Acknowledgement request

A message “M” denotes the kind of its “acknowledgement request” with the element “datagram. header. ackRequest” (see Table 3): The value “true” indicates “acknowledgement message is required” whereas the value “false” indicates “acknowledgement message is NOT required”.

The recipient of the message “M” SHALL submit an acknowledgement message (also referred to as "result message" as described below) (see section 5.2.5.2) if all of the following conditions are fulfilled:

1. The received message “M” belongs to an operation that REQUIRES the evaluation of the acknowledgement request according to section 5.2.4.
2. The received message “M” does not belong to an operation that forbids the evaluation of the acknowledgement request according to section 5.2.4.
3. The received message “M” does not belong to an operation that forbids the creation of an acknowledgement message according to section 5.2.4.
4. The received message “M” indicates “acknowledgement message is required”, i.e. the value of “ackRequest” is set to “true”.

5.2.5.2 Acknowledgement message

An acknowledgement message “N” is related to a received message “M” and indicates whether the received message “M” could be processed or received successfully (positive acknowledgement) or not (negative acknowledgement, i.e. error indication). More precisely, the following scopes of “N” can be assigned:

1. Application success:
   Positive acknowledgement: The recipient of “M” received and evaluated and processed “M” successfully.
2. Application error:
   Negative acknowledgement: The recipient of “M” encountered a problem with “M” or an error occurred with the transmission of “M” to the recipient of “M”.
3. Transmission success:
   Positive acknowledgement: The recipient of “M” confirms it received “M”.
4. Transmission error:
   Negative acknowledgement: An error occurred with the transmission of “M” to the recipient of “M”.

Which scope applies is defined in section 5.2.4.

As the result classifier is used for acknowledgement messages, such messages are also called “result messages”.

A “result message” is composed as follows:

Element "cmdClassifier" SHALL be set to "result". The remaining header elements are set as if the message is a regular reply to the related received message. Furthermore, the element "payload" SHALL contain a "resultData" function as specified by [ResourceSpecification], section “Result”. The element "errorNumber" of “resultData” expresses the kind of the message and SHALL be present and set as follows:

1. A value of "0" SHALL be used for "positive acknowledgement".
2. Every other value denotes a "negative acknowledgement".

In addition, the element “description” of “resultData” MAY be present and filled with a readable information.

In case a “result message” cannot be sent within time according to "defaultMaxResponseDelay" or "maxResponseDelay" described in chapter 5.2.5.3, please refer to chapter 5.2.5.3.

Please note: This section does not specify the circumstances when an acknowledgement message is to be sent or not. For this, please see sections 5.2.4 and 5.2.5.1.

5.2.5.3 Delayed application response

For each feature server a device needs some time to generate a proper response (which could be a message with "reply" classifier or "result" classifier) upon a received request. To have an interoperable concept for devices to know how long to wait for a response, a timeout mechanism with "maximum response delay" is used.

The duration specified by "maximum response delay" only relates to the immediate access to the communication interface of a device. This means that latencies from communication channels are not covered by "maximum response delay".

Note: Each communication channel has some latency. This latency depends on the kind of the communications technology itself as well as on the environment (especially wireless technologies often suffer from poor coverage or too many participants sharing the available bandwidth). As specified above, this kind of latency is independent from the "maximum response delay".

There are two timeout levels to derive the “maximum response delay” for a given server feature:

1. By default, the “maximum response delay” is equal to defaultMaxResponseDelay. The defaultMaxResponseDelay SHALL be 10 seconds.
2. If the maxResponseDelay element within the feature description of the detailed discovery (see section 7.1.2) is set AND the value of the maxResponseDelay element is larger than zero seconds, the value of the maxResponseDelay element SHALL be applied for this feature as “maximum response delay” (i.e. instead of defaultMaxResponseDelay).

Recommendations on the value of maxResponseDelay:

1. If a feature server will usually or frequently need more time than defaultMaxResponseDelay to generate proper replies, it SHOULD set the maxResponseDelay element in the feature description of the detailed discovery (see section 7.1.2) to a duration that the feature server can almost always fall short of for the generation of proper replies.
2. If a feature server will almost always need less time than defaultMaxResponseDelay to generate proper replies, it MAY set the maxResponseDelay element in the feature description of the detailed discovery (see section 7.1.2) to a duration that the feature server can almost always fall short of for the generation of proper replies.

Rules on the use of "maximum response delay":

1. An implementation that complies with this or a subsequent version of the specification SHALL be able to handle response delays of at least defaultMaxResponseDelay.

2. A feature client MAY use "maximum response delay" for the detection of a response-timeout even if "maximum response delay" is shorter than defaultMaxResponseDelay.
3. An implementation SHOULD consider the communications technology specific latency carefully before it begins with the detection of a timeout to an expected response.

5.2.6 Time information in "timestamp"

The header's (optional) element "timestamp" uses the type AbsoluteOrRelativeTimeType. As specified in the document [ResourceSpecification], section "Time information (absolute / relative / recurring)", in case of absolute times the UTC zone shall be applied. A valid example is "2016-04-28T19:43:14.3Z" (in contrast to "2016-04-28T17:43:14.3-02:00" or even the bare local time "2016-04-28T17:43:14.3").

The application of relative times (i.e. according to the type "xs:duration") is of limited use. It is usually only useful for devices that "collect" their data/messages over a certain period and send them only at specific times (e.g. battery powered devices; these often also have no real time clock/UTC setup). In this case, the (negative) relative time is relative to "now" and indicates when the message was created in the past.

5.2.7 Structure

The structure of the SPINE header SHALL be set as shown below.

Element name	Type	M/O/NV /C (see 2.1.1)	Brief explanation
datagram. header		M	The header element SHALL be present.
datagram. header. specificationVersion	SpecificationVersionType (see section 2.2)	M	The version of the SPINE data model applicable to the XML. SHALL be present.
datagram. header. addressSource	FeatureAddressType (see [ResourceSpecification], section "Common data types")	M	The address of the submitter of this XML. SHALL be present.
datagram. header. addressSource. device	AddressDeviceType; see “Device address” in section 7.1.1.2	O	Device part of the source address. SHOULD be present in simple communication mode, SHALL be present in all other communication modes. If the element is present, it SHALL not be empty. See section 5.2.2.1 for details.
datagram. header. addressSource. entity (list)	AddressEntityType (see section 2.2)	M	Entity part(s) of the source address. SHALL be present. See also section 3.2, esp. concerning restrictions on the supported depth.

datagram. header. addressSource. feature	AddressFeatureType (see section 2.2)	M	Feature part of the source address. SHALL be present.
datagram. header. addressDestination	FeatureAddressType (see [ResourceSpecification], section "Common data types")	M	The address of the receiver of this XML. SHALL be present.
datagram. header. addressDestination. device	AddressDeviceType; see "Device address" in section 7.1.1.2	O	Device part of the destination address. SHOULD be present in simple communication mode, SHALL be present in all other communication modes. If the element is present, it SHALL not be empty. See section 5.2.2.1 for details.
datagram. header. addressDestination. entity (list)	AddressEntityType (see section 2.2)	M	Entity part(s) of the destination address. SHALL be present. See also section 3.2, esp. concerning restrictions on the supported depth.
datagram. header. addressDestination. feature	AddressFeatureType (see section 2.2)	M	Feature part of the destination address. SHALL be present.
datagram. header. addressOriginator	FeatureAddressType (see [ResourceSpecification], section "Common data types")	O	Can be used to model information of a "forwarded" SPINE message. The element would contain the address of the original submitter. MAY be present. See also section 5.2.2.2.
datagram. header. addressOriginator. device	AddressDeviceType; see "Device address" in section 7.1.1.2	M	Device part of the originator address. SHALL be present.
datagram. header. addressOriginator. entity (list)	AddressEntityType (see section 2.2)	M	Entity part(s) of the originator address. SHALL be present. See also section 3.2, esp. concerning restrictions on the supported depth.
datagram. header. addressOriginator. feature	AddressFeatureType (see section 2.2)	M	Feature part of the originator address. SHALL be present.
datagram. header. msgCounter	xs:unsignedLong	M	The message number of the submitter of this SPINE message. SHALL be present.
datagram. header. msgCounterReference	xs:unsignedLong	O	The message number of the related SPINE message. SHALL NOT be set if the message does not relate to another message. Otherwise it SHALL be present and set to the message number of the related message.
datagram. header. cmdClassifier	CmdClassifierType, see section 5.2.4	M	The so-called "classifier" associated to the given SPINE function. This denotes for which kind of operation a function

			is used (read, write, notify, etc.). SHALL be present. See section 5.2.4.
datagram. header. ackRequest	xs:boolean	O	Indicates the kind of “acknowledgement request” of the message. The value “true” indicates that an explicit acknowledgement message for this message is requested. May be present. If absent, the default value “false” applies. See section 5.2.5.1 for details.
datagram. header. timestamp	AbsoluteOrRelativeTimeType (see section 2.2)	O	The timestamp of the creation of this SPINE message. May be present. See section 5.2.6.

Table 3: Structure of the SPINE header

5.3 Payload

5.3.1 General information

Within the element “payload”, a single "command" can be placed (broadly speaking; the data model is also prepared for future extensions, but this is not considered further in detail in this version of the specification). It permits or even requires the presence of additional elements to express/identify a functionality. Details are discussed in the subsequent sections.

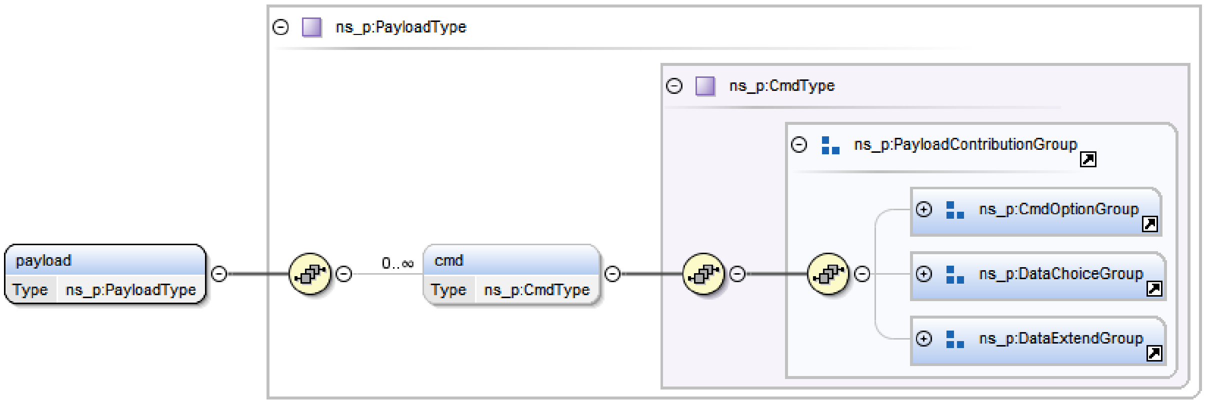


Figure 6: SPINE payload

5.3.2 Elements and usage

Within the protocol specification, the elements of the SPINE payload SHALL be set as shown in Table 4.

Element name	**Type**	**M/O/NV/C (see 2.1.1)**	**Brief explanation**
datagram. payload		M	The payload element SHALL be present.
datagram. payload. cmd		1..unbounded	Each “cmd” instance can take information for one function. Note: In this version of the specification just one (the first) occurrence of “cmd”

			within "payload" SHALL be used (i.e. although the data model permits the occurrence of multiple "cmd" instances in theory, just one is considered).
datagram. payload. cmd. function	FunctionType (see section 2.2)	O	Contains the function name of element "datagram.payload.cmd.<FUNCTION>". SHALL be present if datagram.payload.cmd.filter is present. SHALL be absent otherwise. See section 5.3.4.
datagram. payload. cmd. filter		0..unbounded	Identifies the content that shall be restricted (see section 5.3.4). Although the model permits multiple occurrences, this version of the specification permits two occurrences at maximum. SHALL ONLY be present if at least one child element is present.
datagram. payload. cmd. filter. filterId	xs:unsignedInt	O	Reserved for future use.
datagram. payload. cmd. filter. cmdControl		O	Specifies the kind of function/data restriction. SHALL be present for specified function/data restrictions only. SHALL be absent otherwise. See section 5.3.4. Note: In this version of the specification, exactly one of the possible child elements SHALL be present if datagram.payload.cmd.filter.cmdControl is present.
datagram. payload. cmd. filter. cmdControl. delete		O	Denotes the restriction is a "delete" operation.
datagram. payload. cmd. filter. cmdControl. partial		O	Denotes the restriction is a "partial" operation.
datagram. payload. cmd. filter. <SELECTORS>		0..unbounded	"<SELECTORS>" is a placeholder for SPINE class specific "Selectors" definitions. E.g. for a SPINE function "X" there might be a specific "XSelectors" defined. Please look at the SPINE data model for a list of all possible entries. If present, it SHALL be an instance of the specific "selectors" of the function "datagram. payload. cmd. <FUNCTION>". I.e. with the function name of "...cmd.<FUNCTION>" abbreviated with "X", the "Selectors" name in "<SELECTORS>" SHALL be "XSelectors". Can occur multiple times. All occurrences shall be interpreted as logical "OR" operation.

datagram. payload. cmd. filter. <ELEMENTS>		O	“<ELEMENTS>” is a placeholder for SPINE class specific “Elements” definitions. E.g. for a SPINE function “X” there might be a specific “XElements” defined. Please look at the SPINE data model for a list of all possible entries. If present, it SHALL be an instance of the specific “Elements” of the function “datagram. payload. cmd. <FUNCTION>”. I.e. with the function name of "...cmd.<FUNCTION>" abbreviated with “X”, the “Elements” name in “<ELEMENTS>” SHALL be “XElements”.
datagram. payload. cmd. <FUNCTION>		M	“<FUNCTION>” is a placeholder for exactly one SPINE function. SHALL be present.
datagram. payload. cmd. manufacturerSpecific Extension	xs:hexBinary	O	Can be used to extend a given function with proprietary data. Please note it shall only be used together with a function. MAY be present.
datagram. payload. cmd. lastUpdateAt	AbsoluteOrRel ativeTimeTyp e (see section 2.2)	O	Can be used in case an implementation caches data of the actual node. MAY be present.

Table 4: Elements of the SPINE payload

As already explained in chapter 3, on each feature there SHALL be at maximum one class implemented with regards to the feature’s primary functionality.

The general XSD-based SPINE payload definition permits the presence of multiple “cmd” instances and therefore multiple functions in theory. However, within this version of the protocol specification each instance of a “payload” element SHALL contain exactly one “cmd” instance.

5.3.3 Ownership

The combination of functions and classifiers/operations of the SPINE data model should be applied considering the concept of "ownership". The concept of “ownership” is directly linked to the SPINE role concept. The “ownership” of data is always defined on SPINE feature level. The relation between ownership and role is defined in chapter 3, esp. items 7 and 8.

The "owner" of data (functions) may

- notify its own data,
- reply a request (i.e. received read operation) with its own data,
- update its own data upon request (i.e. upon received write operation).

This, of course, only provided that these actions are supported by the owner.

Especially the write operation should be considered. As example we assume device "A" with a client feature “AA” submits a write operation with data "X" to device "B" with server feature “BB”. Then, data "X" must be considered to be owned and "known" by feature “BB” of device "B". I.e. device "A" should not expect device "B" considers "X" as data owned by feature “AA” of device "A".

Call operations and acknowledgement messages (with classifier “result”) have a different scope, hence do not belong to the concept of ownership.

5.3.4 Restricted function exchange with cmdOptions

5.3.4.1 Overview

The restricted function exchange (RFE) concept is used to exchange just a certain restricted part of a function instead of the full function. The following elements of “datagram.payload.cmd” are subsequently called “cmdOptions”:

1. datagram. payload. cmd. function
2. datagram. payload. cmd. filter
3. datagram. payload. cmd. filter. cmdControl

Note: The cmdOption “datagram. payload. cmd. function” SHALL be used and include the correct function name if and only if any of the other cmdOptions is used.

If full functions are requested or exchanged (“full function exchange”), cmdOptions are NOT used.

Please note the cmdOption “datagram. payload. cmd. function” just serves as some kind of preface in order to introduce which function to operate on in some cases. I.e. it DOES NOT contain function data.

The basic difference between restricted and full functions is determined by the absence or presence of cmdOptions.

Please note: The following subsections show cmdOptions combinations that also contain the case of full function exchange for comparison.

In general, the support of restricted function exchange is optional unless a featureType requires the support explicitly. Even then, the featureType may support only specific kinds of restricted function exchange. Please also consider section 5.3.4.9.

Let's assume "T" is a data model definition (including proper presence indications for each element) for a full function exchange. An example for "T" is the SPINE function "smartEnergyManagementPsData" of the feature type "SmartEnergyManagementPs" as specified in [ResourceSpecification]. Section 5.3.4 defines general requirements of "restricted function exchange" and it defines for specific cases which elements are required at least. For example, a "notify" with "cmdControl" set to "partial" can be used to convey a <FUNCTION> that contains added or modified function parts. In this case added and modified elements are required, whereas unchanged elements (even if the definition of "T" designates them as "mandatory") are not required (unless rules of section 5.3.4 impose specific requirements as in case of identifiers, e.g.). Of course, for each required (child) element its respective parent elements are always mandatory.

Remark: A featureType may specify one or more specifically restricted variants of "T". For example, the feature type "SmartEnergyManagementPs" as specified in [ResourceSpecification] defines the "primary use" of the SPINE function "smartEnergyManagementPsData" to express energy management related information of a device (conveyed in a "read" or "notify" operation). Additionally, the featureType also defines variants of this SPINE function to shift a functionality of the device or select a different option (these variants use "write" operations). These variants can be seen as extracts or simplifications of the "primary use" of the function. They are esp. useful to describe dedicated changes of the device's SPINE function.

On the cardinality of “filter”:

As shown in Table 4, the element “datagram. payload. cmd. filter” may occur more than one time. This cardinality is used to permit a restricted function exchange with two “filter” parts within one message: One “filter” part with sub-element “cmdControl” set to “delete” and another “filter” part with sub-element “cmdControl” set to “partial”. As brief example we take one of the permitted "write" cmdOptions combinations of section 5.3.4.2 where a “delete” as well as a “partial” part are used (functional details on these parts are explained in section 5.3.4.2). We assume some "xyzData" list entries of the function “xyzListData” are deleted and some list entries are partially modified for this example. The “payload” part could then look like this (with some content replaced by “...” to improve readability):

```
...
<payload>
    <cmd>
        <function>xyzListData</function>
        <filter>
            <cmdControl><delete/></cmdControl>
            <xyzListDataSelectors>...</xyzListDataSelectors>
            <xyzDataElements>...</xyzListDataElements>
        </filter>
        <filter>
            <cmdControl><partial/></cmdControl>
            <xyzListDataSelectors>...</xyzListDataSelectors>
        </filter>
        <xyzListData>
            <xyzData>
                ...
            </xyzData>
        </xyzListData>
    </cmd>
</payload>
```

In the following subsections the combinations are explained in tables where each "filter" part is grouped together with the respective child elements in three columns. In Table 5 the first three columns with "cmdControl", "<ELEMENTS>", and "<SELECTORS>" belong to a "filter" element where "cmdControl" is set to "delete". Likewise, the second group (columns 4 to 6) belong to a "filter" element where "cmdControl" is set to "partial". The "payload" example above corresponds to the penultimate row of Table 5.

filter with “delete”	**filter with “partial”**		

cmdControl	<ELEMENTS>	<SELECTORS>	cmdControl	<ELEMENTS>	<SELECTORS>	<FUNCTION>	Explanation and rules
-	-	-	partial	-	-	X	...
...	...	...	...	...	...	...	...
delete	-	X		-	-	dc	...
...	...	...	...	...	...	...	...
delete	(X)	(X)	partial	-	(X)	X	...
-	-	-	-	-	-	X	...

Table 5: Example table (template): This template is used in the subsequent sections for specific cmdOptions combinations. In this template, each "..." is just a placeholder.

5.3.4.2 “write” cmdOptions combinations

The following table shows which kind of cmdOptions combinations can be considered for the use with classifier “write”.

filter with “delete”			filter with “partial”				
cmdControl	**<ELEMENTS>**	**<SELECTORS>**	**cmdControl**	**<ELEMENTS>**	**<SELECTORS>**	**<FUNCTION>**	**Explanation and rules**
-	-	-	partial	-	-	X	<FUNCTION> contains function parts to add or modify. Additionally, <FUNCTION> may restrict the locations (specific list items) by using identifiers, as described in section 5.3.4.6. List items in <FUNCTION> that have NO identifier SHALL be applied to all corresponding list entries of the data owner.
-	-	-	partial	-	X	X	<SELECTORS> specify locations (specific list items), as described in section 5.3.4.7, where <FUNCTION> data is added or modified. <FUNCTION> SHALL NOT use identifiers. <SELECTORS> SHALL match with already existing locations. Therefore, it is not possible to add new list entries with this combination.
delete	-	X		-	-	dc	Locations (specific list items) specified by <SELECTORS> shall be deleted.
delete	X	-		-	-	dc	Elements specified by <ELEMENTS> shall be deleted.
delete	X	X		-	-	dc	Locations (specific list items) and elements that shall be deleted are determined by <SELECTORS> and <ELEMENTS>, according to section 5.3.4.7 and section 5.3.4.8.
delete	(X)	(X)	partial	-	(X)	X	At first the filter with cmdControl "delete" SHALL be applied according to the “delete” combinations described above. Afterwards <FUNCTION> and filter with cmdControl "partial" SHALL be applied according to the “partial” combinations described above.

-	-	-	-	-	-	X	Full write.

Table 6: Considered cmdOptions combinations for classifier "write".

- "X" means a proper instance is present and not empty in the message.
- "(X)" means a proper instance may be present. If present, it is not empty.
- "-" means no such item is in the message.
- "dc" means that the corresponding instance must be present but can be ignored (don't care; i.e. in case of pure "delete" commands (no additional "partial" part) a <FUNCTION> instance must be present but shall be empty).

In this version of the specification, at maximum one "delete" filter and at maximum one "partial" filter SHALL be used in one command. If both filters are present, the "delete" filter SHALL be present before the "partial" filter in the command.

In general, a write operation with restricted function exchange SHALL ONLY be executed by a server if it can execute the received operation completely.

5.3.4.3 "notify" cmdOptions combinations

A "notify" command is very similar constructed like a "write" command, because a notify is mostly used to communicate what has changed after a write process. Be it an external write or internal change, both can be viewed as write processes. Therefore, the "notify" cmdClassifier permits also the same combinations as the "write" cmdClassifier. Please consider the details provided in section 5.3.4.2.

The following table shows which kind of cmdOptions combinations can be considered for the use with classifier "notify".

filter with "delete"			**filter with "partial"**				
cmdControl	**<ELEMENTS>**	**<SELECTORS>**	**cmdControl**	**<ELEMENTS>**	**<SELECTORS>**	**<FUNCTION>**	**Explanation and rules**
-	-	-	partial	-	-	X	<FUNCTION> contains added or modified function parts. Additionally, <FUNCTION> may restrict the locations (specific list items) by using identifiers, as described in section 5.3.4.6. List items in <FUNCTION> that have NO identifier SHALL be applied to all corresponding list entries of the data owner.
-	-	-	partial	-	X	X	<SELECTORS> specify locations (specific list items), as described in section 5.3.4.7, where <FUNCTION> data was added or modified. <FUNCTION> SHALL NOT use identifiers.
delete	-	X		-	-	dc	Locations (specific list items) specified by <SELECTORS> were deleted.
delete	X	-		-	-	dc	Elements specified by <ELEMENTS> were deleted.
delete	X	X		-	-	dc	Locations (specific list items) and elements specified by <SELECTORS> and <ELEMENTS> were deleted, according to section 5.3.4.7 and section 5.3.4.8.

delete	(X)	(X)	partial	-	(X)	X	At first elements specified by filter with cmdControl "delete" were deleted according to the “delete” combinations described above. Afterwards <FUNCTION> and filter with cmdControl "partial" were applied according to the “partial” combinations described above.
-	-	-	-	-	-	X	Full notify.

Table 7: Considered cmdOptions combinations for classifier "notify"

- “X” means a proper instance is present and not empty in the message.
- “(X)” means a proper instance may be present. If present, it is not empty.
- “-” means no such item is in the message.
- “dc” means that the corresponding instance must be present but can be ignored (don’t care; i.e. in case of pure "delete" commands (no additional "partial" part) a <FUNCTION> instance must be present but shall be empty).

In this version of the specification, at maximum one “delete” filter and at maximum one “partial” filter SHALL be used in one command. If both filters are present, the “delete” filter SHALL be specified before the “partial” filter in the command order.

5.3.4.4 “read” cmdOptions combinations

The following table shows which kind of cmdOptions combinations can be considered for the use with classifier “read”.

filter with “partial”				
"cmdControl"	**<SELECTORS>**	**<ELEMENTS>**	**<FUNCTION>**	**Explanation and rules**
partial	X	-	present but EMPTY	Locations (specific list items) specified by <SELECTORS>, as described in section 5.3.4.7, shall be returned.
partial	-	X	present but EMPTY	Elements specified by <ELEMENTS>, as described in section 5.3.4.8, shall be returned.
partial	X	X	present but EMPTY	Locations (specific list items) and elements that shall be returned are determined by <SELECTORS> and <ELEMENTS> according to section 5.3.4.7 and section 5.3.4.8.
-	-	-	-	Full read.

Table 8: Considered cmdOptions combinations for classifier "read"

- “X” means a proper instance is present and not empty in the message.
- “-” means no such item is in the message.

5.3.4.5 "reply" cmdOptions combinations

The following table shows which kind of cmdOptions combinations can be considered for the use with classifier "reply".

filter with "partial"				
"cmdControl"	**<SELECTORS>**	**<ELEMENTS>**	**<FUNCTION>**	**Explanation and rules**
partial	-	-	X	<FUNCTION> contains requested function parts. <FUNCTION> may contain identifiers, as described in section 5.3.4.6. In this case identifiers of each list item in the reply SHALL be full, even if the corresponding "read operation" made use of elements selection with "<ELEMENTS>" but did not specify the elements of the identifier.
-	-	-	-	Full reply.

Table 9: Considered cmdOptions combinations for classifier "reply"

- "X" means a proper instance is present and not empty in the message
- "-" means no such item is in the message.

A server MAY ignore unsupported cmdOption combinations and then replies with more than the requested parts instead. If the server does not support cmdOptions with "read" at all, it SHALL respond with a full reply.

5.3.4.6 <FUNCTION> identifiers - Implicit list item selection

Identifiers are used to select specific list entries directly in the <FUNCTION> (element "datagram.payload.cmd.<FUNCTION>" of Table 4). The approach requires that list items can be identified in a unique way. Identifiers for each featureType are defined in [ResourceSpecification].

If a featureType specifies identifiers and their use, then the values of identifiers in a (full or restricted) instance of "<FUNCTION>" serve implicitly for the identification of the proper list item.

The read request belonging to this example can be found in section 5.3.4.7. As requested, device "A" responds with this XML:

```
<datagram>
    <header>
        ...
        <cmdClassifier>reply</cmdClassifier>
    </header>
    <payload>
        <cmd>
            <function>measurementListData</function>
            <filter>
                <cmdControl>
                    <partial/>
                </cmdControl>
```

```
                </filter>
                <measurementListData>
                    <measurementData>
                        <measurementId>5</measurementId>
                        <valueType>minValue</valueType>
                        <timestamp>2015-07-14T15:00:00.0Z</timestamp>
                        <value>
                            <number>-173</number>
                            <scale>-1</scale>
                        </value>
                        <evaluationPeriod>
                            <startTime>2015-07-14T10:00:00.0Z</startTime>
                            <endTime>2015-07-14T15:00:00.0Z</endTime>
                        </evaluationPeriod>
                        <valueSource>measuredValue</valueSource>
                        <valueTendency>rising</valueTendency>
                        <valueState>normal</valueState>
                    </measurementData>
                </measurementListData>
            </cmd>
        </payload>
    </datagram>
```

In this example it is assumed the device keeps no further history of “minValue” values, i.e. it keeps just the latest “minValue”.

The following rules apply if cmdOptions are used:

1. <FUNCTION> identifiers SHALL only be used for “partial” write/notify/reply commands.
2. If <FUNCTION> identifiers are used, it is not allowed to use <SELECTORS> in a filter with cmdContol “partial”.
3. If <FUNCTION> contains at least one identifier, ALL list items included in <FUNCTION> SHALL have complete identifiers, as specified in section 5.3.4.6.1. I.e. it is not valid to have a list item without identifier and another list item with identifier within a <FUNCTION> instance.

5.3.4.6.1 Identifier hierarchy and completeness of list identifiers

SPINE class functions with multiple identifiers usually assign the identifiers a “natural hierarchy” (e.g. an identifier “xId” serves as some kind of “parent identifier” for an identifier “yId”; a common example could be a postal address where the identifier "city" is a "parent identifier" of the identifier "street"). A featureType can explicitly define which identifiers are used and which hierarchy applies. If the featureType does not specify identifiers, the default identifiers of the underlying class apply and the hierarchy follows the order stated in the identifier section of the class description.

The identifiers of a list item are complete if no identifier needs to be added to identify the target list item uniquely.

Among others, identifiers permit "finding" or choosing specific list entries of a function containing a large list. An example is shown in section 5.3.4.7.

5.3.4.7 <SELECTORS> – Explicit list item selection

This section describes the use of element “datagram.payload.cmd.filter.<SELECTORS>” of Table 4.

5.3.4.7.1 Common rules and description

<SELECTORS> are used to select specific list entries by referencing a value of a certain child element from the corresponding list entry. One possibility is to select list entries by one or more identifiers. Some <SELECTORS> also provide other search criteria.

All list entries, where the child element has the corresponding value, are selected. This means, the list entry and all the child elements of the list entry are selected. Usage of <ELEMENTS> allows restricting a selection further.

The featureType specifies which values can be used for <SELECTORS> and what elements are exactly selected when a match was found.

In this version of the specification, list structures can occur in the following functions:

1. In list-based SPINE standard class functions ("measurementListData", e.g.). All SPINE class functions of the pattern "xListData" are list-based SPINE standard class functions.
2. In SPINE complex class functions with internal list structures. Some functions of the complex class "SmartEnergyManagementPs" have internal list structures, e.g. as complex functions are composed of basic functions (or at least their types), the selectors of complex functions are composed of (one or more) selectors (or at least their types) of list-based standard class functions.

<SELECTORS> are mostly needed for filter with cmdControl "delete" and for "partial" read commands. However, <SELECTORS> also allow "partial" writing/notifying the same values in/for multiple list entries at the same time.

For many resources default values are defined for certain elements. If such elements are omitted, the default value applies. If a client explicitly sets an element with its default as selector but the server has ommitted the corresponding element, the server SHOULD still succesfully match the corresponding selector element value against the default value.

A brief example for a "read operation" shall be given: In this example we assume device "A" is the owner of a feature with a "measurementListData" instance. Device "B" wants to get a copy of this instance where only such "measurementData" list items are embedded that have the element "measurementId" set to "5" and "valueType" set to "minValue". Thus, device "B" sends the following read operation:

```
<datagram>
    <header>
        ...
        <cmdClassifier>read</cmdClassifier>
    </header>
    <payload>
        <cmd>
            <function>measurementListData</function>
            <filter>
                <cmdControl>
                    <partial/>
                </cmdControl>
                <measurementListDataSelectors>
                    <measurementId>5</measurementId>
                    <valueType>minValue</valueType>
                </measurementListDataSelectors>
```

```
                </filter>
                <measurementListData/>
            </cmd>
        </payload>
    </datagram>
```

It can be seen that the “read” request with the empty function “measurementListData” is accompanied with the dedicated selectors “measurementListDataSelectors” and proper values.

5.3.4.7.2 Selectors with address elements

Some selectors can take address elements to select a specific "device" address part or "entity" address part or "feature" address part. In fact, for such selectors parts the same rules as of section 5.3.4.7.1 apply. However, esp. due to the intrinsic list structure of "entity" address parts it shall be emphasized that still only exact matches of an "entity" address part of a selectors with an "entity" address part within a function are concsidered as valid matches. An example shall illustrate this:

We consider an extract of the address tree example from section 3.2. The tree begins with the device address part "someDevice":

```
“someDevice”
...
      +--- entity 1
      |        +--- entity 4 (child of “someDevice”/entity 1)
      |        |        +--- feature 7 (*1)
...
      +--- entity 4           (child of “someDevice”)
      |        +--- feature 1 (child of “someDevice”/entity 4)

(*1): child of “someDevice”/entity 1/entity 4
```

From this tree some address paths are shown with their XML representation in the subsequent table. The "someDevice" device address part above is simplified for brevity and – according to the permitted pattern – shown in full length in these XML examples:

No.	Address path	XML representation
A1	device “someDevice” / entity 1	<device>d:_i:46925_someDevice</device>
A2	device “someDevice” / entity 1	<device>d:_i:46925_someDevice</device> <entity>1</entity>
A3	device “someDevice” / entity 1 / entity 4	<device>d:_i:46925_someDevice</device> <entity>1</entity> <entity>4</entity>
A4	device “someDevice” / entity 1 / entity 4 / feature 7	<device>d:_i:46925_someDevice</device> <entity>1</entity> <entity>4</entity> <feature>7</feature>
A5	device “someDevice” / entity 4	<device>d:_i:46925_someDevice</device> <entity>4</entity>
A6	device “someDevice” / entity 4 / feature 1	<device>d:_i:46925_someDevice</device> <entity>4</entity> <feature>1</feature>

Table 10: Address path examples

Now we consider an extract of a selectors (in this case of "nodeManagementDetailedDiscoveryDataSelectors") with an "entityAddress" part comprising of the optional child element "device" and an optional list of "entity" items:

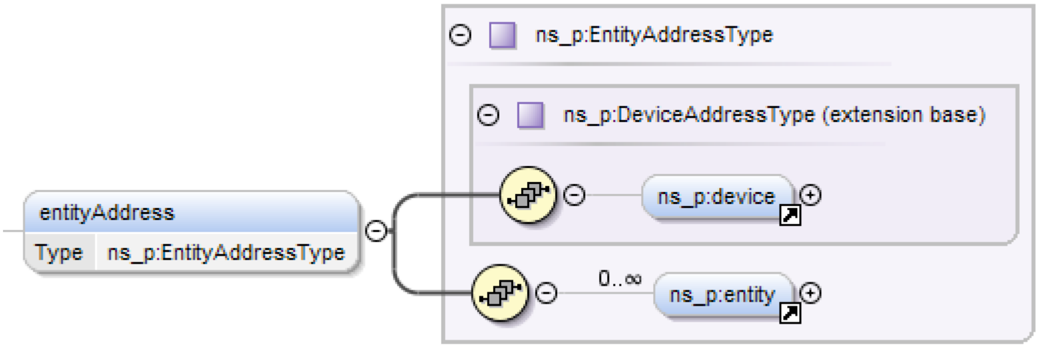


Figure 7: Example of selectors part (extract) with entity address part

A selectors instance like

```
<entityAddress>
      <entity>1</entity>
      <entity>4</entity>
</entityAddress>
```

would match address parts A3 and A4 of Table 10.

A selectors instance like

```
<entityAddress>
      <entity>4</entity>
</entityAddress>
```

would match address parts A5 and A6 of Table 10, but neither A3 nor A4.

5.3.4.8 <ELEMENTS> - Selection of "elements"

Each function (element "datagram.payload.cmd.<FUNCTION>" of Table 4) is defined with specific (usually optional) child elements. The <ELEMENTS> definition (element "datagram.payload.cmd.filter.<ELEMENTS>" of Table 4) includes all elements from <FUNCTION> but without type and value. Within the "<ELEMENTS>" definition it is possible to tell explicitly which subset of a function's elements is to be considered.

Note: <ELEMENTS> need not include identifiers in most cases.

<ELEMENTS> SHALL only be used in the following two cases:

- Data deletion (write/notify)
- "partial" read

An "<ELEMENTS>" instance contains no values that can be used for identification of certain list items. However, the application needs to consider as well if certain list items are selected explicitly via <SELECTORS> (section 5.3.4.7).

Subsequently we iterate through all elements of a given "<ELEMENTS>" instance and call the iterated element "<element>". The following rules determine whether <element> shall be applied to the corresponding element of the target function and (in case of list items) to which list item of the target function it shall be applied:

1. If <SELECTORS> is NOT used in the command:
   <element> is applied.
2. If <SELECTORS> is used in the command:
   <element> is applied only for list items selected via <SELECTORS>.

Among others this means that the restricted function exchange may denote specific list items and the “<ELEMENTS>” content then shall only be applied to those list items and not to other list items of the data owner.

5.3.4.9 Minimum restricted function exchange support

Esp. for “basic functions” (standard featureTypes with list-based standard class functions, e.g.) it is useful to define whether and how a certain kind of minimum support for restricted function exchange can be achieved. Therefore, a feature that supports such kind of minimum restricted function exchange SHALL indicate this by stating the "partial" flag in the "possibleOperations" for "read" or "write" (or both) for a respective function (see section 7.1.1.5.5). However, the application of this support and further rules that need to be considered are specified in [ResourceSpecification], section "Restricted function exchange for list-based functions".

6 Communication modes

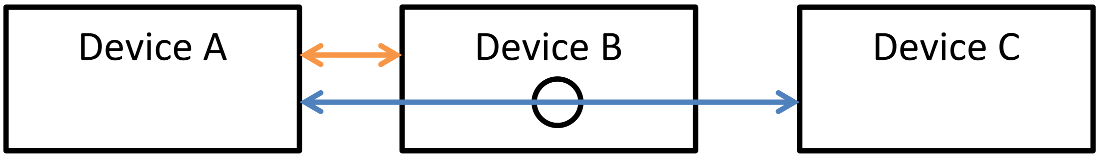


Figure 8: Communication modes of SPINE devices A, B and C. The circle in device B symbolises the "message forwarding" task of device B.

In Figure 8 different communication modes are briefly depicted:

1. An orange arrow shows the simple communication mode which assumes the direct communication between two SPINE devices.
2. A blue arrow shows the enhanced communication mode, which provides the possibility to send messages via intermediate devices, as needed by a (SPINE) technology gateway. In Figure 8 messages between devices “A” and “C” are exchanged via device “B”.

The modes are only used to distinguish the kind of the connection or message flow according to Figure 8. For each mode different requirements apply. This permits some simplification with regards to the use of address fields if SPINE devices are "directly connected" and only need to exchange data belonging to the respective communication partner.

The simple communication mode SHALL be supported by every SPINE device.

The enhanced communication mode SHALL be supported by all SPINE devices that have their networkFeatureSet element (see section 7.1.1.5.3) set to a value different than "simple" (e.g. “smart” or “gateway” or “router”).

If two devices exchange messages directly (without any device "between" them) the simple communication mode applies. If two devices need (at least) a third one to communicate with each other, the enhanced communication mode applies.

Please note that the communication mode may change with every message. I.e. with regards to Figure 8, device "A" may exchange messages for/from device "B" in simple communication mode, while (at almost the same time frame) messages for/from device "C" pass device "B" as well but with enhanced communication mode.

6.1 Simple communication mode

In this mode, two SPINE devices “A” and “B” communicate with the following restrictions:

1. The devices “A” and “B” are considered having a direct connection, i.e. there is no third SPINE device “C” in the communication between devices “A” and “B”.
2. In datagrams exchanged between devices “A” and “B” the source and destination address elements of the datagram’s header SHALL NOT contain any other address information than those of device “A” or “B”. I.e. there is no address information of any other device.
3. The devices “A” and “B” SHALL expect that payload exchanged between these devices belongs to device “A” or “B” only. I.e. they SHALL not expect to convey information of any third device.

In simple communication mode, the device part of the addresses (source and destination) SHOULD be set in order to easily identify the communication partners. Item 3 above describes how received messages have to be interpreted if the sender did not set a device address part in the addressSource or addressDestination element.

Advice: A device SHOULD NOT be implemented in a way that it supports the simple communication mode ONLY. Instead, a device SHOULD support the enhanced communication mode as well. Simply put, a device SHOULD be implemented in a way that its own networkFeatureSet (see section 7.1.1.5.3) can be assigned a DIFFERENT value than "simple".

Please consider also section 7.1.1.5.3.

6.2 Enhanced communication mode

The enhanced communication mode can only apply if both communication partners (source and destination) have their networkFeatureSet element (see section 7.1.1.5.3) set to a value different than "simple". Of course, all devices that are involved in the communication as intermediate device (hop / forwarding device) must have a networkFeatureSet value different than "simple", too.

The enhanced communication mode provides the possibility to send messages via an intermediate SPINE device. This situation already occurs for technological gateways, i.e. SPINE-capable devices that bring devices of other communications technologies into the SPINE world by representing them with an own device address within a SPINE network (in this example the gateway derives/assigns a SPINE device address for the device of the other communications technology; this is of course technology dependent and therefore not defined in detail in this document; please consider [TechnologyMappings] for such details). These “native” devices can only be accessed via the SPINE gateway. An informative example of enhanced communication mode and DestinationList can be found in Annex E. Additionally, this kind of “forwarding” is rather generic and can be extended to SPINE devices which are no technological gateways, but that are also not just “simple”. Among others, this applies to devices with networkFeatureSet set to “smart”.

If a SPINE device supports the simple as well as the enhanced communication mode, it SHALL set its own networkFeatureSet property to a value different than "simple" (e.g. “smart” or “gateway” or “router”). See section 7.1.1.5.3 for details.

If a SPINE device “X” itself can act as intermediate device, this means basically it is capable of forwarding a received message to another SPINE device (provided that it knows how to access both devices). I.e. device “X” may receive a message with a SPINE source “device” address of device “Y” and a SPINE destination “device” address of device “Z”. Such kind of message forwarding REQUIRES that device “X” supports “enhanced communication mode” AND that it has a proper “DestinationList” implemented (see section 7.2).

Note: The networkFeatureSet values of the SPINE devices “X”, “Y”, and “Z” do not need to be identical.

The device part of all addresses SHALL be set when using the enhanced communication mode to identify where the message originates from and where it shall be routed to.

7 Functional commissioning

Functional commissioning comprises mechanisms like binding and subscription (but not any commissioning mechanisms of underlying communications technologies or layers). These mechanisms require the definition and assignment of roles with regards to a dedicated functionality (feature). Altogether this leads to clear responsibilities between devices and helps to implement rather automatic exchange of information between devices.

In order to support binding and subscription a discovery mechanism is defined that SHOULD be executed in advance.

The focus on binding and subscription is primarily on the definition of “ownership” of data. The owners SHALL push information updates to dedicated recipients (further operations are permitted but skipped right now for convenience).

Besides this approach to push information it is still possible to pull information (i.e. read specific information from the owner of the data) without a previously configured binding or subscription. However, the latter might be manufacturer specific and is not considered further in this chapter unless for cases explicitly described.

The subsequent sections describe messages (more precise: message parts) and rules on NodeManagement instances. The messages are described in tables, more details are described in the SPINE data model XSD definition (EEBus_SPINE_TS_NodeManagement.xsd).

The message parts of the subsequent sections only show the SPINE function specific payload part of the element “datagram. payload”.

7.1 Detailed discovery

Every device has some functionality which it can announce to other devices via the discovery mechanisms of SPINE.

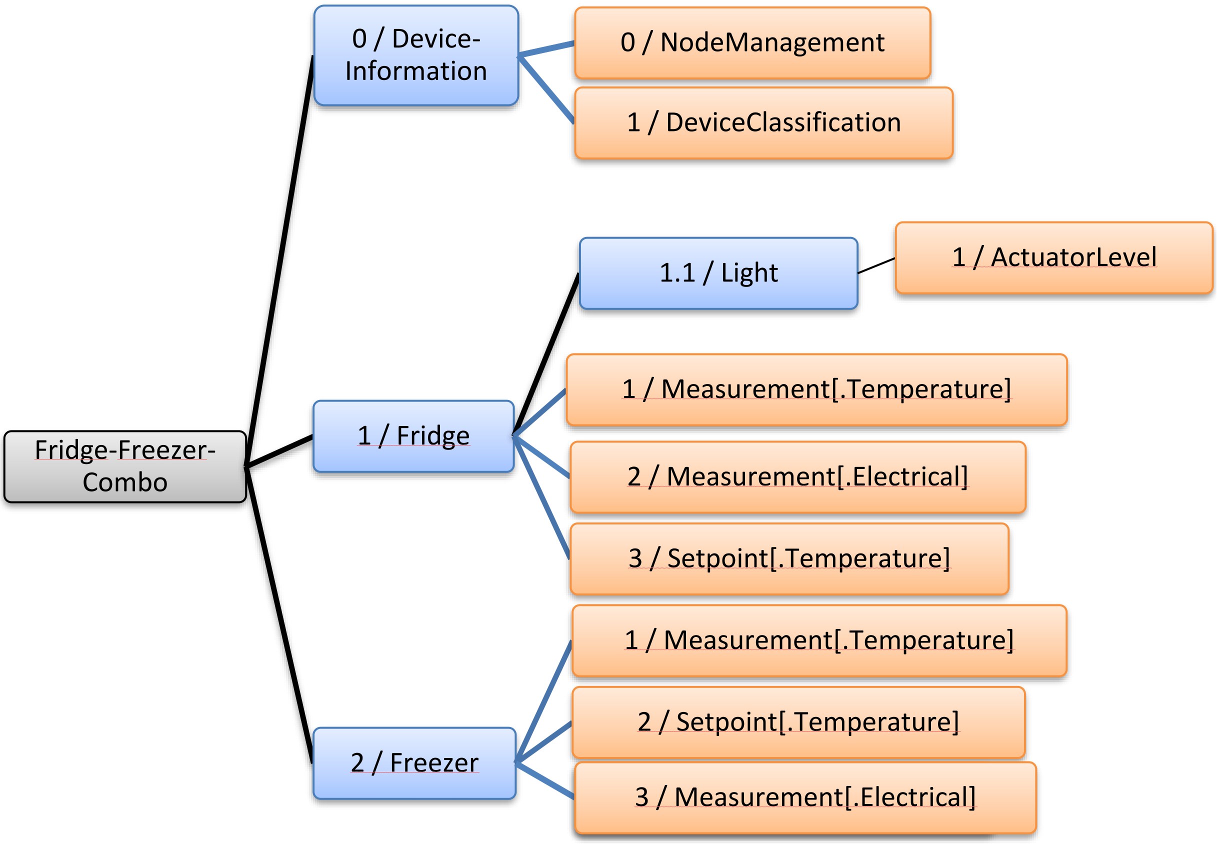


Figure 9: Discovery example

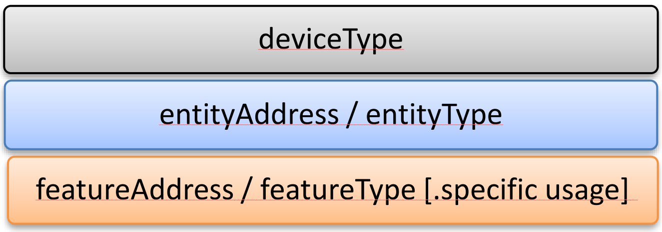


Figure 10: Hierarchy types. Entities can contain child-entities; "entityAddress" contains all "entity" parts starting from the respective root entity.

The general architecture was already explained in chapter 3 and will now be used in the context of the discovery process. In this specification, a physical device is represented by a SPINE device with a *deviceType*, the SPINE device is subdivided into entities with an *entityType* and features with a *featureType*.

1. A SPINE device holds a collection of one or more entities. The **deviceType** describes a generic context (e.g. "Washer"), but may also describe a certain minimal set of entities that can be expected on this device.
2. An entity holds a collection of one or more (child) entities and features. The **entityType** describes a generic context (e.g. "Fridge"), but may also describe a certain minimal set of (sub-)entities and features that can be expected on this entity. A special entityType "DeviceInformation" gives information on the device itself (see also section 3).
3. A feature with the role "server" or "special" holds a collection of one or more functions. The **featureType** describes a generic context (e.g. "ActuatorLevel"), but may also describe a certain minimal set of functions that can be expected on this feature. The role expresses that

the feature is “owner” of the functions (except for “...Call” functions that are only used with the "call" classifier).

4. A feature with the role “client” may as well keep a list of functions and express its featureType. However, features of this role do not “own” the functions (i.e. they cannot be read on such a feature, among others). They can just “use” features of the role “server” or special. In the subsequent discussions the role “client” is not considered in detail.
5. A function holds a collection of one or more elements. The elements might have simple or complex data types. All readable or writeable functions can be discovered over a function list in the feature description. The information on the device, entities and features are provided by the primaryNodeManagement instance of each SPINE device and can be discovered during a detailed discovery process (see below).
6. A read command on the function provides a copy of the function with its elements (the copy might be restricted dependent on the kind of the read command).

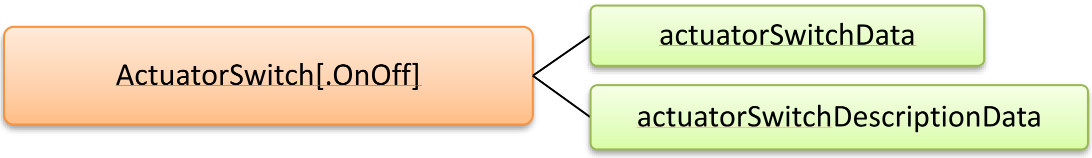


Figure 11: Function Discovery Example over Feature Description

7.1.1 Basic definitions and rules

Detailed discovery requires the presence of own NodeManagement instances (presentation of own information) and the access to remote NodeManagement instances (gather information of another device). Detailed discovery SHALL be supported by all SPINE devices on their entity 0 (entity with entity address = 0) and feature 0 (feature with feature address = 0).

A SPINE node discovers the remote SPINE node’s device information, entity information and feature information using the mechanisms specified in this section. Detailed discovery between trusted SPINE nodes (see Annex B) may be triggered at any time. Section 7.1.3 discusses the situation if a primary NodeManagement instance alters its device information, the set of entities or features during runtime and how this SHALL be handled.

In general, a SPINE node performs a detailed discovery of another SPINE node because it is searching for features (or more specifically for featureTypes) to connect to. As an example, a node which has an “ActuatorSwitch” client searches for an “ActuatorSwitch” server to connect to. If a client finds a fitting feature it is recommended to use this feature “in time” (e.g. send a “read” request) as explained in Annex D.

Note: A client does not need to implement a specific client feature like "ActuatorSwitch" (from the example above) to connect to an "ActuatorSwitch" server feature. Instead, a client may handle all its functionality with one feature, although that may cover different featureTypes on the server side.

A SPINE feature type may be specialized by a specific usage. As an example, a “Measurement” server could be specialized to say “Temperature”, which means it measures temperatures. Standardized specific usages can be found in the [ResourceSpecification].

For mandatory rules about creating and deleting bindings and subscriptions, consider sections 7.3 and 7.4 respectively.

7.1.1.1 Rules for vendor specific extensions

Some types (but not all) may be extended by the vendor (see [ResourceSpecification] for details), e.g. the MeasurementType may be extended by some values (special measurement types) that are not explicitly listed in the standardized enumeration. Device types, entity types and feature types may be vendor specific, too.

For extensible string-based types a vendor specific extension SHALL fulfil the following pattern (notated as XML schema regular expression):

```
_(i:[1-9][0-9]*|n:[a-zA-Z0-9-]+)_[^\p{Cc}\p{Cf}\p{Z}]+
```

The first underscore introduces that the value is not a standardized value. The marker "i:" introduces an IANA PEN, the marker "n:" introduces a name of the vendor. The IANA PEN SHOULD be used! The expression beyond the second underscore permits the use of Unicode characters EXCEPT for such Unicode characters belonging to the so-called "general category" definitions "Control", "Format", and "Separator". Among others, this means white space characters are not permitted.

Remark: IANA PENs can be requested for free at [IANA PEN] and are unique. The requirement for uniqueness can typically not be achieved easily with “name based” identifications as this would usually require as well a central authority for the reservation of names.

Note: The underlying XSDs of SPINE may be used within other specifications and standards that are out of the responsibility of the EEBus Initiative e.V. Therefore, we added the possibility to use the vendor name instead of the IANA PEN to the regular expression stated above.

Note: The IANA PEN SHALL not be used for visualizing a brand name to the consumer (instead "DeviceClassification" SHOULD be used, see [ResourceSpecification]).

7.1.1.2 Rules for devices

The description of a (physical) device mainly consists of some general information and the list of provided entities. Manufacturers are free to choose an entity design for their SPINE node, consisting of standardized entity types and proprietary entity types.

Device type:

Devices are usually specified by a device type (like “Washer”, e.g.). Some device types are published by the EEBus Initiative e.V. But due to the large number of different types of devices, many vendors will specify their own device type. In either case the device type is modelled in the deviceType element of "deviceInformation. description". The specification provides some standardized values for the device type. In case of vendor specific device types section 7.1.1.1 SHALL be applied for the value.

The device type SHOULD NOT be used for any kind of automatism.

Device address:

The address of a device is modelled as a string. Its length is restricted to a maximum of 256 characters. Up to this revision of SPINE, only the following pattern is permitted for the device address string (notated as XML schema regular expression):

```
d:_(i:[1-9][0-9]*|n:[a-zA-Z0-9-]+)_[^\p{Cc}\p{Cf}\p{Z}]+
```

The pattern requires the address to first state "d:", then the pattern of the vendor specific extensions (see section 7.1.1.1) follows to give the address the needed uniqueness. After that, the vendor states a string containing a vendor-wide unique address that MAY consist of the following characters: Any Unicode character EXCEPT for such Unicode characters belonging to the so-called "general category" definitions "Control", "Format", and "Separator". Among others, this means white space characters are not permitted.

Examples for possible device strings:

- d:_i:46925_ABCabc-123
- d:_i:46925_0123456789

The IANA PEN SHOULD be used! The device address part after the second underline SHALL be unique! Each vendor is responsible for the uniqueness of its device addresses.

Note: The underlying XSDs of SPINE MAY be used within other specifications and standards that are out of the responsibility of the EEBus Initiative e.V. Therefore, we added the possibility to use the vendor name instead of the IANA PEN to the regular expression stated above.

Note: The IANA PEN SHOULD not be used for visualizing a brand name to the consumer (instead "DeviceClassification" SHOULD be used, see [ResourceSpecification]).

7.1.1.3 Rules for entities

A standardized entity type has a set of standardized features and child-entities, which are either optional or mandatory.

In general, each entity (standardized or non-standardized) can hold a mixture of:

1. Child-entities (see section 3.2).
2. Features which are included in the standardized entity type.

A SPINE node SHALL NOT expect other SPINE nodes to have any other features in a standardized entity type but the ones included in the standardized entity type. This means the other node may have additional features on the entity but it is not valid to insist on the presence of these additional features just from the standardized entity type. I.e. other included features of the remote node have to be discovered manually.

Please note the architecture described in this document requires a specific mandatory SPINE entity at least: The entity containing the so-called “primary NodeManagement instance”. This entity has the entityType “DeviceInformation” as described in section 3.

Entity types used in a SPINE node, which do not follow any standardized entity descriptions, SHALL be marked as vendor specific (see section 7.1.1.1 for details).

A standardized entity is a SPINE node that implements a standardized entity type and has or has not been extended with non-standardized content.

7.1.1.4 Rules for features

A feature that uses a standardized featureType in a standardized way SHALL NOT be marked as vendor specific (see section 7.1.1.1 for details). A feature which uses a proprietary class SHALL NOT use a standardized feature type and SHALL be marked as vendor specific. A feature which uses a standardized class but uses a proprietary feature type SHALL be marked as vendor specific.

7.1.1.5 Rules for specific element usage

The subsequent sections discuss some elements that are used in the “detailed discovery” messages (see section 7.1.2).

7.1.1.5.1 Usage of element “deviceAddress. device”

According to section 5.2.2 the “device” information elements of two devices SHALL be different. Section 7.1 defines different messages for “discovery” processes between NodeManagement instances of two devices “A” and “B”. Some of these messages contain the element “deviceInformation. description. deviceAddress. device”. This element conveys the originator’s “device” address part.

The requirement on the uniqueness of the “device” part of the address REQUIRES a device “A” to terminate a connection with a communication partner (device “B”) immediately if device “A” recognizes that its own “device” value is identical to the one of device “B”. Similarly, device “B” SHALL terminate a connection with device “A” if it detects the non-uniqueness of “device”.

7.1.1.5.2 Usage of element networkManagementResponsibleAddress

The element “networkManagementResponsibleAddress” is reserved for specific network management purposes that will be defined in a later version of this specification.

7.1.1.5.3 Usage of element networkFeatureSet

The value “simple” applies implicitly (i.e. as default value) in case the element “networkFeatureSet” is not present. A device that supports only the “simple communication mode” SHALL NOT use any other value than “simple” for its own value of “networkFeatureSet”.

Further values for “networkFeatureSet” are defined by the SPINE data model. In this case the element “networkFeatureSet” SHALL be present.

1. If a device supports the “enhanced communication mode” (i.e. it can send/receive via intermediate SPINE devices) it should set its “networkFeatureSet” to “smart” unless one of the following conditions is more appropriate.
   Remark: A “smart” device may also act as “intermediate device” as explained in the following items. But “smart” devices are expected to rather offer or use specific functionalities (features) than to just forward messages.
2. If – in addition to the previous item – a device primarily acts as “intermediate device” (i.e. its primary purpose is to forward SPINE messages according to the given destination address; see below for details) it should set its “networkFeatureSet” to “router” unless the following condition is more appropriate.
3. If – in addition to the previous items – a device primarily acts as technology gateway it should set its “networkFeatureSet” to “gateway”.

Note: Further specific functionalities of "router" and "gateway" remain subject of future versions of the specification.

Whether the “simple communication mode” or the “enhanced communication mode” applies is described in detail in chapter 6. Esp. section 6.2 explains why “enhanced communication mode” is required for “intermediate devices”.

7.1.1.5.4 Usage of element minimumTrustLevel

This element is used to report whether the owner (i.e. “server”) of a specified information element (more precisely: a specified address) requires a minimum “trust level” in order to permit functional access (see also Annex B). This means a client needs to gain at least this trust level towards the server in order to access the information of the specified address. The purpose of the element minimumTrustLevel is to reduce the number of unsuccessful access attempts because of insufficient access rights. However, the evaluation of minimumTrustLevel is technology dependent. Therefore, only some general aspects can be described in this document.

A “trust level” is a rather abstract information. In general, it may consist of distinct components that need to be fulfilled to grant unlimited access to an address. The protocol specification can be used over different communications technologies. Each technology defines own mechanisms to establish a connection between two devices. These mechanisms may already include the assignment of a trust level. On the other hand, minimum trust levels may be unknown in various cases (proprietary implementations or devices bridged from other technologies). Therefore, only some basic rules on the use of minimumTrustLevel can be given:

1. The absence of the element minimumTrustLevel denotes an unknown minimum trust level.
2. The element minimumTrustLevel shall be applied per address level.
3. In this version of the specification the element minimumTrustLevel is used only on feature level (future versions of this specification may use minimumTrustLevel on other address levels than feature as well).

7.1.1.5.5 Usage of element possibleOperations

A feature with the role “server” (or, in some cases, "special") can report its supported functions and some operation details (see “featureInformation. featureDescription. supportedFunction” of the function “nodeManagementDetailedDiscoveryData”). The element “possibleOperations” can be used to denote specific operation details granted by a server for the given function towards a client. This is described subsequently.

If a server feature's function supports being read (and can send appropriate replies) the child element “read” SHALL be present. This denotes the support of full function exchange with read-reply operations. If the server feature's function also supports restricted function exchange for a read-reply operation, the sub-element “partial” SHALL be present as well (please distinguish this “partial” from other elements of this name; see note below). However, in a first step this denotes just a general support of restricted function exchange. I.e. a server is NOT obliged to support every kind of restriction requested by a client. In general, a server is permitted to discard unsupported cmdOptions combinations and data, and reply with a less restricted function instead. Please note that featureType specifications may state which kind of restrictions should or shall be supported for read-reply operations. Please consider esp. section 5.3.4.9.

If a server feature's function supports being written the child element “write” SHALL be present. This denotes the support of full function exchange with write operations (i.e. the full replacement of the server’s data of this function). If the server feature's function also supports restricted function exchange for a write operation, the sub-element “partial” SHALL be present as well (please distinguish this “partial” from other elements of this name; see note below). However, in a first step this denotes just a general support of restricted function exchange. I.e. a server is NOT obliged to support every kind of restriction written by a client. In general, a server is permitted to reject unsupported cmdOptions combinations and data completely. Please note that some featureType specifications state which kind of restrictions should or shall be supported for write operations. Please consider esp. section 5.3.4.9.

Note: The sub-element “partial” in “possibleOperations” SHALL NOT be confused with the “cmdControl” element “partial”!

Please note that this version of the specification does not specify child elements of “possibleOperations” for other operations (notify, call).

7.1.2 Detailed discovery “all at once”

The request for the device information, all entities and all attached features SHALL be sent using a message with a “read” classifier from a source node address of the own primary NodeManagement instance (i.e. entity 0, feature 0) and to a destination node address of the recipient’s primary NodeManagement instance (i.e. entity 0, feature 0). The content of payload SHALL be an empty function “nodeManagementDetailedDiscoveryData” (i.e. it SHALL NOT have any child element) of the complex class “NodeManagement”.

A primary NodeManagement instance of a device that receives this request SHALL create a response according to the usual rules (e.g. set the classifier to “reply”, set message number elements (msgCounter, msgCounterReference) accordingly, etc.). The content of payload of this reply SHALL conform to the function “nodeManagementDetailedDiscoveryData” of the complex class “NodeManagement” and SHALL be used as shown below.

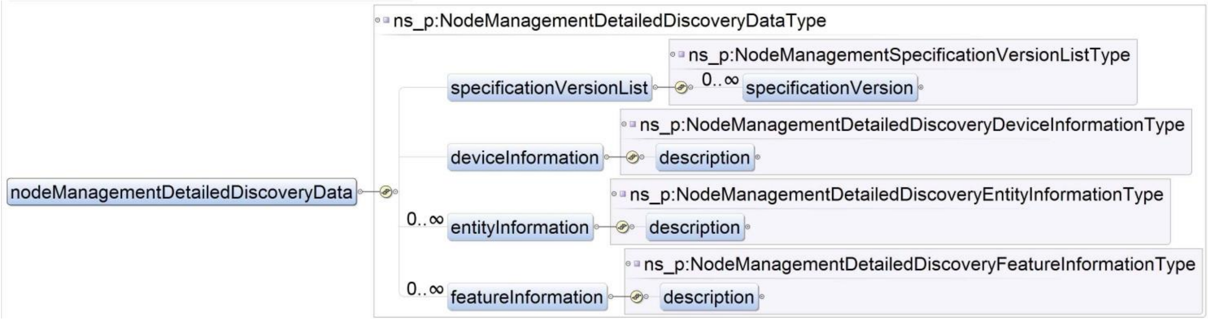


Figure 12: nodeManagementDetailedDiscoveryData function overview, part 1

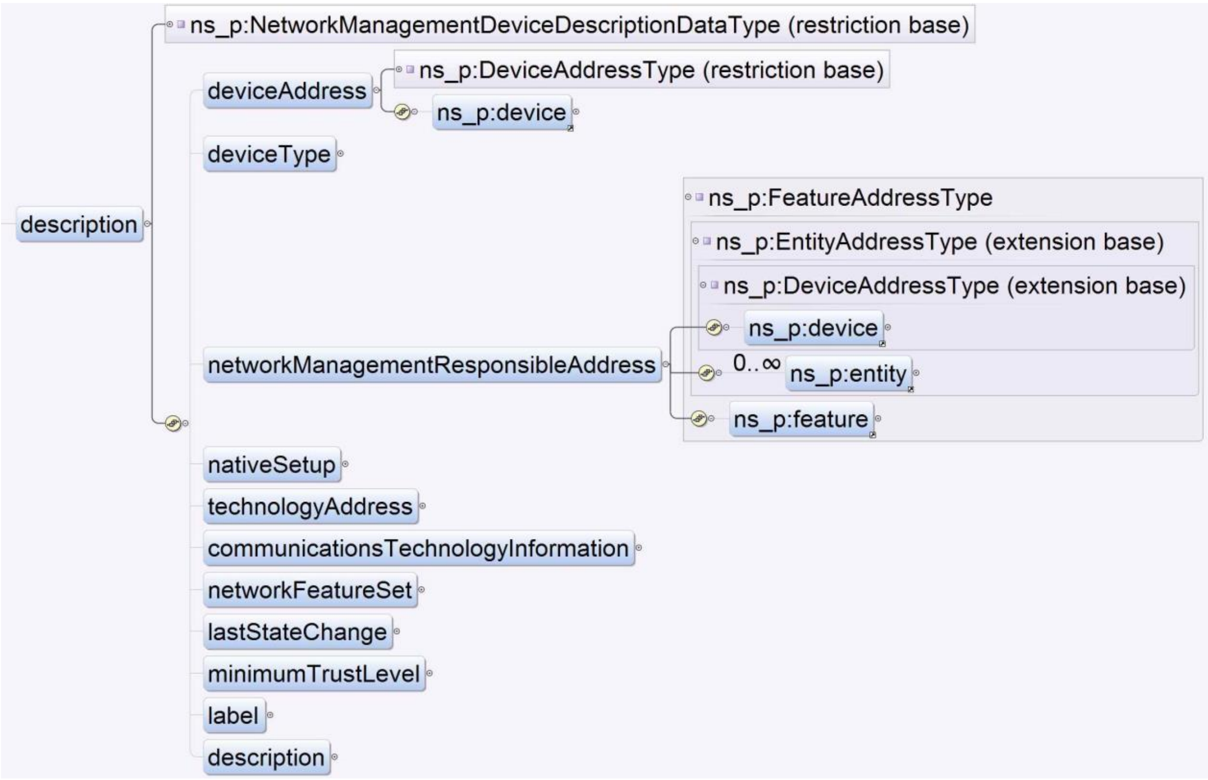


Figure 13: nodeManagementDetailedDiscoveryData function overview, part 2: deviceInformation.description

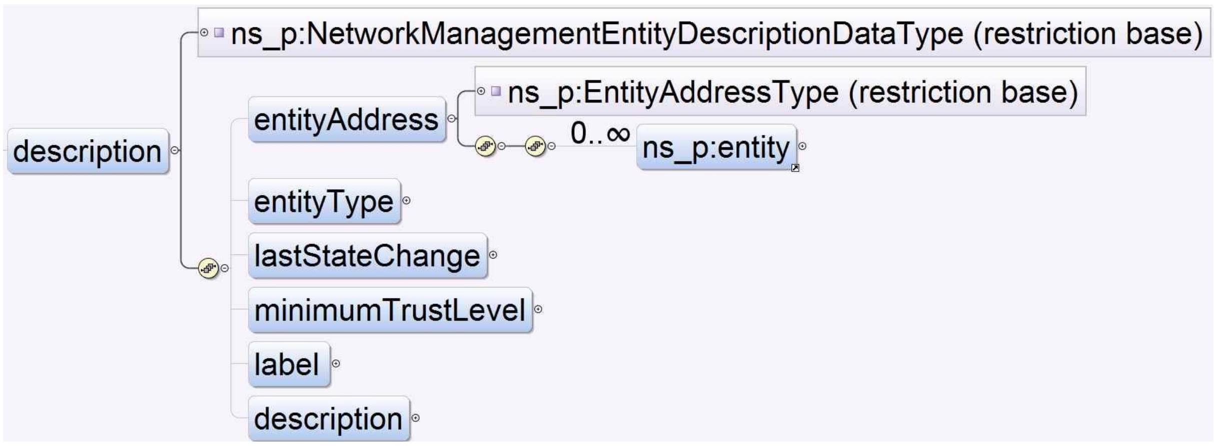


Figure 14: nodeManagementDetailedDiscoveryData function overview, part 3: entityInformation.description

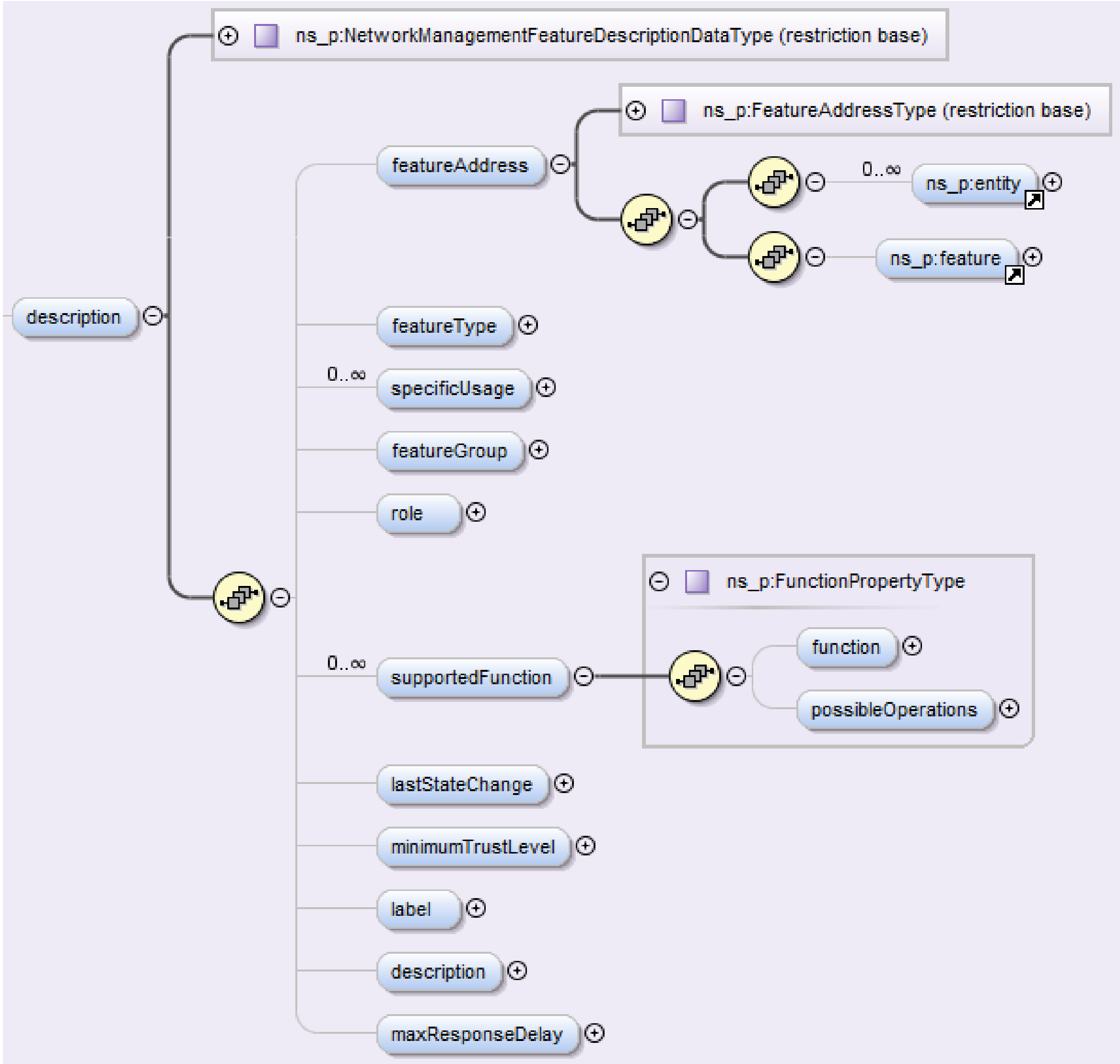


Figure 15: nodeManagementDetailedDiscoveryData function overview, part 4: featureInformation.description

Element name	Type	M/O/NV/C (see 2.1.1)	Explanation
specificationVersionList		M	SHALL always be present and contain a list of specificationVersion.
specificationVersionList. specificationVersion (list)	SpecificationVersionType (see section 2.2)	1..unbounded	SHALL always be present (at least once) and state the SPINE specification versions supported by the device (e.g. 1.0.0 and 1.5.3). Subsequent to the detailed discovery process, two devices SHALL use the highest version supported by both partners.
deviceInformation		M	SHALL be present.
deviceInformation. description		M	SHALL be present

deviceInformation. description. deviceAddress		M	SHALL be present and hold the device address information.
deviceInformation. description. deviceAddress. device	AddressDeviceType; see “Device address” in section 7.1.1.2	M	SHALL be present and hold the device address string.
deviceInformation. description. deviceType	DeviceTypeType (see section 2.2)	M	SHALL be present to denote the type of device. For further rules about the deviceType, please consider section 7.1.1.2.
deviceInformation. description. networkManagementResponsibleAddress		O	Reserved for future use (the address of the “network management responsible” for the whole device is only used for specific network management purposes, see section 7.1.1.5.2 for details).
deviceInformation. description. networkManagementResponsibleAddress. device	AddressDeviceType; see “Device address” in section 7.1.1.2	O	The device address part of this devices’ “network management responsible device”.
deviceInformation.description. networkManagementResponsibleAddress. entity (list)	AddressEntityType (see section 2.2)	0..unbounded	The entity address part(s) of this devices’ “network management responsible device”.
deviceInformation. description. networkManagementResponsibleAddress. feature	AddressFeatureType (see section 2.2)	O	The feature address part of this devices’ “network management responsible device”.
deviceInformation. description. nativeSetup	NetworkManagementNativeSetupType	O	MAY be present and state the technology dependent or implementation dependent configuration to identify (and maybe configure) the device. The string-length SHOULD NOT be longer than 512 characters. If it is longer, the sender SHALL consider the possibility that the receiver will shorten the string to 512 characters.
deviceInformation. description. technologyAddress	NetworkManagementTechnologyAddressType	O	MAY be present and state the address, the device has in its own communications technology. The string-length SHOULD NOT be longer than 512 characters. If it is longer, the sender SHALL consider the possibility that the receiver will shorten the string to 512 characters.

deviceInformation. description. communicationsTechnologyInformation	NetworkManagementCommunicationsTechnologyInformationType	O	MAY be present and state the technology dependent or implementation dependent information on the device with focus on communications aspects. The string-length SHOULD NOT be longer than 512 characters. If it is longer, the sender SHALL consider the possibility that the receiver will shorten the string to 512 characters.
deviceInformation. description. networkFeatureSet	NetworkManagementFeatureSetType (see section 2.2)	O	MAY be omitted in case of “simple”. SHALL be present otherwise. Please consider section 7.1.1.5.3 for details.
deviceInformation. description. lastStateChange	NetworkManagementStateChangeType (see section 2.2)	O	MAY be used to denote the last change on the device (added, removed or modified)
deviceInformation. description. minimumTrustLevel	NetworkManagementMinimumTrustLevelType	O	Reserved for future use. Consider section 7.1.1.5.4 for details. The string-length SHOULD NOT be longer than 64 characters. If it is longer, the sender SHALL consider the possibility that the receiver will shorten the string to 64 characters.
deviceInformation. description. label	*Common data type "LabelType". See section 2.2.*	O	MAY be present. Manufacturers may assign their devices a brief label. This makes interfacing, e.g. to a smartphone application, a lot easier. However, the content of the “label” element is not standardized. The string-length SHOULD NOT be longer than 256 characters. If it is longer, the sender SHALL consider the possibility that the receiver will shorten the string to 256 characters.
deviceInformation. description. description	*Common data type "DescriptionType". See section 2.2.*	O	MAY be present. In a case where a manufacturer wants to present a textual description for a device, he may use this field. However, the content of the “description” element is not standardized. The string-length SHOULD NOT be longer than 4096 characters. If it is longer, the sender SHALL consider the possibility that the

			receiver will shorten the string to 4096 characters.
entityInformation (list)		1..unbounded	SHALL be present. Each occurrence of entityInformation contains information of a specific entity.
entityInformation. description		M	SHALL be present.
entityInformation. description. entityAddress		M	SHALL be present.
entityInformation. description. entityAddress.entity (list)	AddressEntityType (see section 2.2)	1..unbounded	SHALL be present and state the entity address part(s)
entityInformation. description. entityType	EntityTypeType (see section 2.2)	M	SHALL be present and state the entity type of this entity. The entity type can be a standardized one or can be manufacturer specific.
entityInformation. description. lastStateChange	NetworkManagementStateChangeType (see section 2.2)	O	MAY be used to denote the last change on the entity (added, removed or modified)
entityInformation. description. minimumTrustLevel	NetworkManagementMinimumTrustLevelType	O	Reserved for future use. Consider section 7.1.1.5.4 for details. The string-length SHOULD NOT be longer than 64 characters. If it is longer, the sender SHALL consider the possibility that the receiver will shorten the string to 64 characters.
entityInformation. description. label	*Common data type "LabelType". See section 2.2.*	O	MAY be present. Manufacturers may assign their devices a brief label. This makes interfacing, e.g. to a smartphone application, a lot easier. However, the content of the “label” element is not standardized. The string-length SHOULD NOT be longer than 256 characters. If it is longer, the sender SHALL consider the possibility that the receiver will shorten the string to 256 characters.
entityInformation. description. description	*Common data type "DescriptionType". See section 2.2.*	O	MAY be present. In a case where a manufacturer wants to present a textual description for a device, he may use this field. However, the content of the “description” element is not standardized. The string-length SHOULD NOT be longer than 4096 characters. If it

			is longer, the sender SHALL consider the possibility that the receiver will shorten the string to 4096 characters.
featureInformation (list)		0/1..unbounded	SHALL be present for all features that have the role "server" or "special". MAY be present for all features with role "client". Each occurrence of featureInformation contains information on a specific feature.
featureInformation. description		M	SHALL be present.
featureInformation. description. featureAddress		M	SHALL be present.
featureInformation. description. featureAddress. entity (list)	AddressEntityType (see section 2.2)	1..unbounded	SHALL be present and state the entity address part(s)
featureInformation. description. featureAddress. feature	AddressFeatureType (see section 2.2)	M	SHALL be present and state the feature address part
featureInformation. description. featureType	FeatureTypeType (see section 2.2)	M	SHALL be present and state the feature type of this feature. The feature type can be a standardized one or can be manufacturer specific.
featureInformation. description. specificUsage (list)	FeatureSpecificUsageType	0..unbounded	Deprecated
featureInformation. description. featureGroup	FeatureGroupType (see section 2.2)	O	If one or more features are specifically related to each other, they SHOULD use the same feature group number (e.g. "#2"). See also document [ResourceSpecification], section “Feature Group”. Its length is restricted to a maximum of 128 characters.
featureInformation. description. role	RoleType (see section 2.2)	M	Dependent on the functional role this element SHALL be set to “client” or “server” or “special”.
featureInformation. description. supportedFunction (list)		0/1..unbounded	SHALL be present in case of “server” or “special” role. Each occurrence contains information on a function that is supported on this feature.
featureInformation. description. supportedFunction. function	FunctionType (see section 2.2)	M	SHALL be present. Contains the name of the supported function.
featureInformation. description.	PossibleOperationsType (see section 2.2)	O	SHALL be present if one or more child elements are set. Otherwise it SHALL be omitted. Specifies a

supportedFunction. possibleOperations			server's (or server-like in case of "special") supported operations. See section 7.1.1.5.5 for details.
featureInformation. description. supportedFunction. possibleOperations. read		O	SHALL be present if the feature supports receiving "read" commands for the given function and sends appropriate replies. SHALL be absent otherwise. See section 7.1.1.5.5 for details.
featureInformation. description. supportedFunction. possibleOperations. read. partial		O	SHALL be present if the feature supports "restricted function exchange" with read operations (see section 5.3.4.9 for minimum requirements). SHALL be absent otherwise. See section 7.1.1.5.5 for details. Note: The sub-element "partial" in "possibleOperations" SHALL NOT be confused with the "cmdControl" element "partial"!
featureInformation. description. supportedFunction. possibleOperations. write		O	SHALL be present if the feature supports receiving "write" commands for the given function. SHALL be absent otherwise. See section 7.1.1.5.5 for details.
featureInformation. description. supportedFunction. possibleOperations. write. partial		O	SHALL be present if the feature supports "restricted function exchange" with write operations (see section 5.3.4.9 for minimum requirements). SHALL be absent otherwise. See section 7.1.1.5.5 for details. Note: The sub-element "partial" in "possibleOperations" SHALL NOT be confused with the "cmdControl" element "partial"!
featureInformation. description. lastStateChange	NetworkManagementStateChangeType (see section 2.2)	O	MAY be used to denote the last change on the feature (added, removed or modified)
featureInformation. description. minimumTrustLevel	NetworkManagementMinimumTrustLevelType	O	Reserved for future use. Consider section 7.1.1.5.4 for details. The string-length SHOULD NOT be longer than 64 characters. If it is longer, the sender SHALL consider the possibility that the receiver will shorten the string to 64 characters.
featureInformation. description. label	*Common data type "LabelType".*	O	MAY be present. Manufacturers may assign their devices a brief label. This makes interfacing, e.g.

	See section 2.2.		to a smartphone application, a lot easier. However, the content of the “label” element is not standardized. The string-length SHOULD NOT be longer than 256 characters. If it is longer, the sender SHALL consider the possibility that the receiver will shorten the string to 256 characters.
featureInformation. description. description	*Common data type "DescriptionType". See section 2.2.*	O	MAY be present. In a case where a manufacturer wants to present a textual description for a device, he may use this field. However, the content of the “description” element is not standardized. The string-length SHOULD NOT be longer than 4096 characters. If it is longer, the sender SHALL consider the possibility that the receiver will shorten the string to 4096 characters.
featureInformation. description. maxResponseDelay	MaxResponse DelayType (see section 2.2)	O	Specifies a maximum response time of this feature. MAY be present. If present, the value SHALL contain a duration larger than zero seconds. If the element is absent or its value is zero seconds or smaller, the value of defaultMaxResponseDelay SHALL be applied instead. See section 5.2.5.3 for details.

Table 11: Notify/response list of entities and their corresponding features with nodeManagementDetailedDiscoveryData

7.1.3 Partial Detailed Discovery

In most cases a client will be only interested to discover matching data of a server. In this case a partial read can be performed on the nodeManagementDetailedDiscoveryData function. Please refer to the chapter 5.3.4 for more details on partial read.

Within the filter of the partial read the nodeManagementDetailedDiscoveryDataSelectors allows to exactly specify which device-, entity- or featureType or which device-, entity- or featureAddress is of interest.

The following parameters can be set within the Selectors (please consider esp. section 5.3.4.7.2):

Element name	Explanation
deviceInformation	Holds information of the device
deviceInformation. deviceAddress	Holds the device address that serves as unique identifier
deviceInformation. deviceType	Holds the device type
entityInformation	Holds information of an entity of a device

entityInformation. entityAddress	Holds the entity address on the device
entityInformation. entityType	Holds the entity type
featureInformation	Holds information of a feature of an entity
featureInformation. featureAddress	Holds the feature address on the device
featureInformation. featureType	Holds the feature type

Table 12: nodeManagementDetailedDiscoveryDataSelectors

The top-level elements of the Selectors only apply to the corresponding top-level elements of the function nodeManagementDetailedDiscoveryData:

- The Selectors branch "deviceInformation" only applies to the branch "deviceInformation" of the function nodeManagementDetailedDiscoveryData.
- The Selectors branch "entityInformation" only applies to the branch "entityInformation" of the function nodeManagementDetailedDiscoveryData.
- The Selectors branch "featureInformation" only applies to the branch "featureInformation" of the function nodeManagementDetailedDiscoveryData.

The “featureType” and “entityType” will probably be the Selectors most commonly used within partial detailed discovery and SHALL be supported if partial discovery is supported. However, they SHOULD NOT be used together, as the result would be an empty reply (this is due to the structure of the nodeManagementDetailedDiscoveryData function and the filter logic of the selectors).

7.1.4 Using detailed discovery for automatisms (informative)

Detailed discovery enables a SPINE node A to find out which functionality is offered by SPINE node B and vice versa. Typically, a detailed discovery is performed to establish meaningful binding or subscription subsequently. However, difficulties might arise to automatically determine whether a binding or subscription is “meaningful” (from a user perspective) or not.

The process in which a SPINE node derives meaningful bindings or subscriptions by analysing a retrieved detailed discovery is not described in this specification. This specification only states mandatory rules relevant for such processes, such as which bindings or subscriptions shall or shall not be established. See section 7.3 and 7.4 for these mandatory rules. Depending on the kind of device a manufacturer builds, the process of analysing detailed discovery information and deriving suitable bindings or subscriptions may differ significantly. A manufacturer of a very generic on/off switch for example will most likely only consider the minimum requirements.

SPINE nodes are strongly encouraged to consider a remote node’s entityType, featureType and supported functions in case of an automatic binding/subscription, before performing binding or subscribing. A rather intelligent implementation considering all relevant information can drastically reduce the number of possibly unintentional bindings and subscriptions in the field. However, it is not mandatory to consider the type of a feature to establish a binding or subscription.

Furthermore, each feature is associated to a single entity. An entity comes with an entityType, which describes the purpose of an entity. As an example, a real (physical) device like a combination of a fridge and freezer could be divided into two entities with the entity types: “fridge” and “freezer”.

SPINE nodes MAY evaluate the entityTypes of a node they have just executed a detailed discovery on. However, special care should be taken, because the list of entityTypes will be extended with subsequent specification versions. At the same time, new entityTypes might reuse old featureTypes. Thus, filtering for known entityTypes and discarding the rest of the discovered entityTypes leads to potential problems regarding future extensions of the SPINE specification.

The benefit of filtering by known entityTypes can be very different depending on the type of device. For very generic devices like an on/off switch that can be used to switch on or off virtually anything switchable, filtering by entityTypes SHOULD not be considered. However, filtering by entityType can be feasible for more specialized devices.

Complex SPINE nodes might offer multiple features with the same featureType on different feature addresses. Thus, creating an automatic and precise binding and/or subscription (from a user point of view) from the correct feature of SPINE node A to the correct feature of SPINE node B might be difficult. The evaluation of the according entityType and featureType SHOULD be used to overcome those challenges, if possible. A commissioning tool however, MAY provide specific user-based input to create correct bindings and/or subscriptions.

7.1.5 Changes during runtime

During runtime, a device MAY add/remove/modify one or more of its entities or features. In such case the device SHALL send a proper information update according to Table 11 ("nodeManagementDetailedDiscoveryData" from the primary NodeManagement instance, see section 7.1.2) and according to the following rules:

1. The message SHALL be sent with a "notify" classifier.
2. The message SHOULD be sent as "restricted function exchange".
3. Unchanged entities/features SHOULD NOT be put into the message.
4. If an entity itself is removed:
   a. The element "entityInformation. description. lastStateChange" SHALL be set to "removed".
   b. There SHALL NOT be any "featureInformation" item for this entity.
   c. It is not required to notify a full data removal from the owner's "nodeManagementDetailedDiscoveryData" feature. I.e. the above-mentioned notification of "lastStateChange" with "removed" is sufficient.
5. If an entity is added:
   a. The element "entityInformation. description. lastStateChange" SHALL be set to "added".
   b. All of its "featureInformation" items SHALL be in the message.
6. If an entity is modified:
   a. The element "entityInformation. description. lastStateChange" SHALL be set to "modified".
   b. Each added/removed/modified feature of the entity SHALL be in the message with its element "featureInformation. description. lastStateChange" set properly.

7.2 Destination list

7.2.1 Introduction

In contrast to mere direct connections and information – i.e. two devices "A" and "B" share and convey only information they own, but no information of any third device "C" – the "DestinationList" provides information which "other devices" are accessible. More precisely, a device that has an own

"DestinationList" instance containing address information on "other devices" expresses this way that it forwards messages to these devices (which requires use of the "enhanced communication mode" to enable message forwarding). Dependent on the availability and functionality of devices with own "DestinationList" instance, this may even permit gaining and modelling a full overview on all devices that constitute a local network of SPINE devices.

7.2.2 Architecture requirements

The following rules apply:

1. All rules on the use of NodeManagement as specified in previous and following sections apply.
2. The complex class “NodeManagement” includes the optional “DestinationList” functions. The corresponding functions may be present in the supportedFunction list of a feature with the featureType “NodeManagement”.
3. Usage of DestinationList functions in the “NodeManagement” feature is optional in general. However, there are dependencies with the device’s “networkFeatureSet” of its primary NodeManagement instance. These dependencies are described below and SHALL be considered.

With regards to “networkFeatureSet” the following rules apply: If a device’s “networkFeatureSet” of its primary NodeManagement instance

1. is set to “simple” or absent: DestinationList functions SHOULD NOT be implemented.
2. is set to “smart”: DestinationList functions MAY be implemented.
3. is set to “router”: DestinationList functions SHALL be implemented.
4. is set to “gateway”: DestinationList functions SHALL be implemented.

Note: The rules above only describe the architecture. Further details and rules (e.g. process rules) are discussed separately.

7.2.3 Rules

7.2.3.1 Rules for devices

A device MAY always put information about itself into its own DestinationList.

A device MAY put information about connected “simple devices” into its own DestinationList. A “simple device” is a device that is reported by the DestinationList as device with the element networkFeatureSet set to “simple”.

When a device offers a DestinationList instance it is required to forward messages to the devices it has listed in its DestinationList. Under certain circumstances it may be necessary to modify the payload before forwarding or to take over particular management tasks: We assume device "A" has an own DestinationList instance that contains at least the devices "B" and "C". We also assume device "A" receives an "enhanced communication mode" message from device "B" with destination address set to device "C":

1. If device "C" is capable of "enhanced communication mode", then device "A" SHALL forward the received message unchanged to device "C".
2. If device "C" is NOT capable of "enhanced communication mode" (i.e. it is a "simple device"), then device "A" SHALL act as "SPINE proxy" to device "C" with regards to this message and a potential response message from device "C". This means:
    a. If the received message is neither a binding and nor a subscription message, device "A" forwards the received payload from device "B" in a (usually new) message to device "C". It also means a proper response from device "C" is used by device "A" to send a proper response to device "B". More details on this SPINE proxy concept can be found in section E.5.
    b. If the received message is a binding or subscription message, device "A" needs to become the binding or subscription partner towards device "C" (unless device "C" does not support binding or subscription, resp.). This means device "A" needs to take over binding and subscription management tasks towards device "B".

Remarks:

1. Please note there is no general requirement to put "simple" devices in the own DestinationList instance. However, if this is done it includes the responsibility for SPINE proxy functionality as explained above.
2. It is recommended that a technology gateway puts directly connected devices into its own DestinationList instance if it intends to forward messages to/from these devices. However, there is no general requirement to put all connected devices into the DestinationList instance.

7.2.3.2 Rules for specific element usage

7.2.3.2.1 Usage of element deviceAddress. device

DestinationList information MAY contain information on “simple” devices. Such devices MAY have limited support of the address information element “device” due to the fact that they only support the simple communication mode (see section 6.1). However, the element “device” SHALL be present and unique in each node’s DestinationList segment. Otherwise, the device could not be distinguished or identified.

7.2.3.2.2 Usage of element networkFeatureSet

The values of “networkFeatureSet” from a NodeManagement instance (i.e. from device discovery) and from DestinationList SHALL NOT contradict each other. This means the values must be identical, with the exception that element absence and the value “simple” are considered being equal.

7.2.4 Exchanging DestinationList

7.2.4.1 Requesting DestinationList

The request for a device's DestinationList SHALL be sent using a message with a "read" classifier and a destination node address of the recipient's primary NodeManagement feature. The content of payload SHALL be an "empty" function "nodeManagementDestinationListData", i.e. it SHALL NOT have any child element.

A SPINE node that receives this request SHALL create a response according to the usual rules (e.g. set the classifier to "reply", set message number elements (msgCounter, msgCounterReference) accordingly, etc.). The content of payload of this reply SHALL conform to the function "nodeManagementDestinationListData", as described in the following table.

Figure 16: nodeManagementDestinationListData function overview, part 1

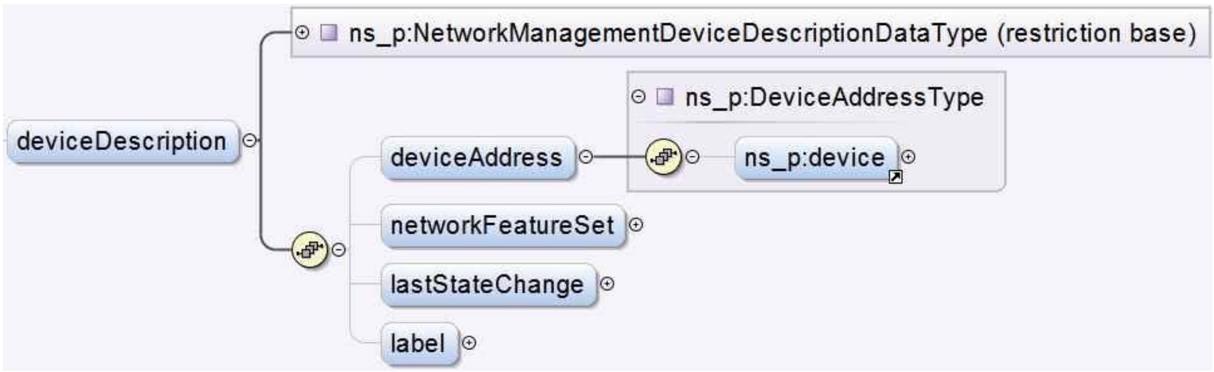


Figure 17: nodeManagementDestinationListData function overview, part 2

Element name	**Type**	**M/O/NV/C (see 2.1.1)**	**Explanation**
nodeManagementDestinationData (list)		1..unbounded	SHALL be present. Each occurrence contains information on one destination.
nodeManagementDestinationData. deviceDescription		M	SHALL be present.
nodeManagementDestinationData. deviceDescription. deviceAddress		M	SHALL be present.
nodeManagementDestinationData. deviceDescription. deviceAddress. device	AddressDeviceType; see "Device address" in section 7.1.1.2	M	SHALL be present and state the device address.
nodeManagementDestinationData. deviceDescription. networkFeatureSet	NetworkManagementFeatureSetType (see section 2.2)	O	MAY be omitted. Absence of this element denotes the default value "simple".

nodeManagementDestinationData. deviceDescription. lastStateChange	NetworkManagementStateChangeType (see section 2.2)	O	MAY be used to denote the last change on the node in DestinationList: Absence of the element denotes that the node is still present, which is equivalent to the value “added”.
nodeManagementDestinationData. deviceDescription. label	*Common data type "LabelType". See section 2.2.*	O	MAY be present. Manufacturers may assign their devices a brief label. This makes interfacing, e.g. to a smartphone application, a lot easier. However, the content of the “label” element is not standardized. The string-length SHOULD NOT be longer than 256 characters. If it is longer, the sender SHALL consider the possibility that the receiver will shorten the string to 256 characters.

Table 13: Notify/response of DestinationList information with nodeManagementDestinationListData

7.2.4.2 Notification of DestinationList

For the notification of DestinationList information the same message as described in section 7.2.4.1, Table 13, SHALL be used, but with the classifier set to “notify” and as “restricted function exchange”. Furthermore, the execution of notifications and the content SHALL be restricted as follows:

1. Only the “nodeManagementDestinationData” array elements that have changed SHOULD be put into a notification.

7.3 Binding

Some standardized SPINE feature types make use of the binding concept and specify concrete responsibilities and permissions (hence, such specific details cannot be given in this section).

Please note that a bound feature client will not automatically be notified about changes of the bound feature server. To receive notifications, the feature client has to subscribe to the feature server (see section 7.4).

7.3.1 Basic definitions and rules

As currently only the binding on a whole feature is possible the term binding always relates to a full feature binding within this specification.

Binding SHALL be supported by a SPINE node

1. if it has at least one feature server that REQUIRES binding for specific tasks, or
2. if it has at least one feature client that needs to use a feature server that REQUIRES binding for specific tasks.

Please note that some feature types define requirements for binding! The "binding" concept specific functions SHALL be implemented on the primary NodeManagement instance.

Establishing a binding requires knowledge of the communication partner's server features. Thus, it is recommended to get this information from a discovery procedure as described in section 7.1. Of course, if during detailed discovery no features to bind to could be found, no binding will be established.

A binding is a special functional relation between one feature with the role "server" (called "feature server") and one feature with the role "client" (called "feature client"). In addition, a feature with a role "special" can be considered as "feature client" or "feature server" (or both) as described below and can then be used for a binding accordingly. Bindings usually include clear responsibilities and permissions between the "feature server" and the "feature client(s)". The concrete responsibilities and permissions are given by standardized featureTypes, hence cannot be specified in this section.

Regarding binding in general the following rules can be described:

1. Dependent on the featureType a feature server MAY limit the number of bindings it permits on a given feature. I.e. the server MAY permit only one (single binding) or multiple bindings on the feature.
2. The request to establish a binding SHALL originate from the node with the "feature client". It SHALL NOT originate from the node with the "feature server".
3. The node with the "feature server" MAY deny a binding request. Remark: This could happen if the server already has a binding with another node on the same feature and no further binding is allowed on the same feature (as explained earlier), or if the trust level requirements are not fulfilled.
4. Binding information SHOULD be kept persistently by both binding partners unless the binding shall be released explicitly.
5. It is not possible to create a binding to entities or devices. It is only possible to create a binding to a feature.
6. Either binding partner is permitted to release the binding.
7. In general, the binding concept also permits to model binding relations between features of the same device (i.e. device-internal bindings). In this case, the rules on absent "device" address parts (see note at the end) SHALL NOT be used to distinguish between "feature client" and "feature server". Rather, the roles of the features need to be used and considered explicitly in order to model a unique relation. Please note that this is not possible if two different features of the same device both have the role "special".
   Note on rules on absent "device" address parts: The following sections specify some functions with child elements "... clientAddress. device" and "... serverAddress. device" (see Table 14, Table 15, Table 16, e.g.; the parent elements are abbreviated here with "..."). These elements can contain device address parts. The explanations in the sections or function definitions contain rules on the absence of these elements.

A feature with the role "special" may have own data or it may consume other device's data (or both). If a "special" feature sends own data (i.e. send reply or notify) or receives a "write" operation to change its own data it acts as "feature server". If a "special" feature receives data owned by another device (i.e. receive reply or notify) or sends a "write" operation to change data it acts as "feature client".

The major benefit of a binding is based upon above mentioned clear rules and responsibilities. As an example, a typical binding is established between a wall mounted light switch acting as “ActuatorSwitch” client and a ceiling light acting as “ActuatorSwitch” server (of course we also assume a proper process of manually pairing the light switch to the light for this example; note that pairing processes are beyond the scope of this specification). Once the binding is established, the switch knows where to send a “toggle” command to and the light knows from whom to accept a “toggle” command.

7.3.2 Create Binding

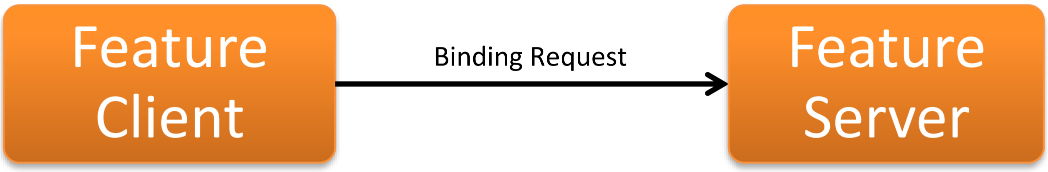


Figure 18: Binding request

A nodeManagementBindingRequestCall SHALL always be sent using a “call” classifier. The content of payload is described in the following table.

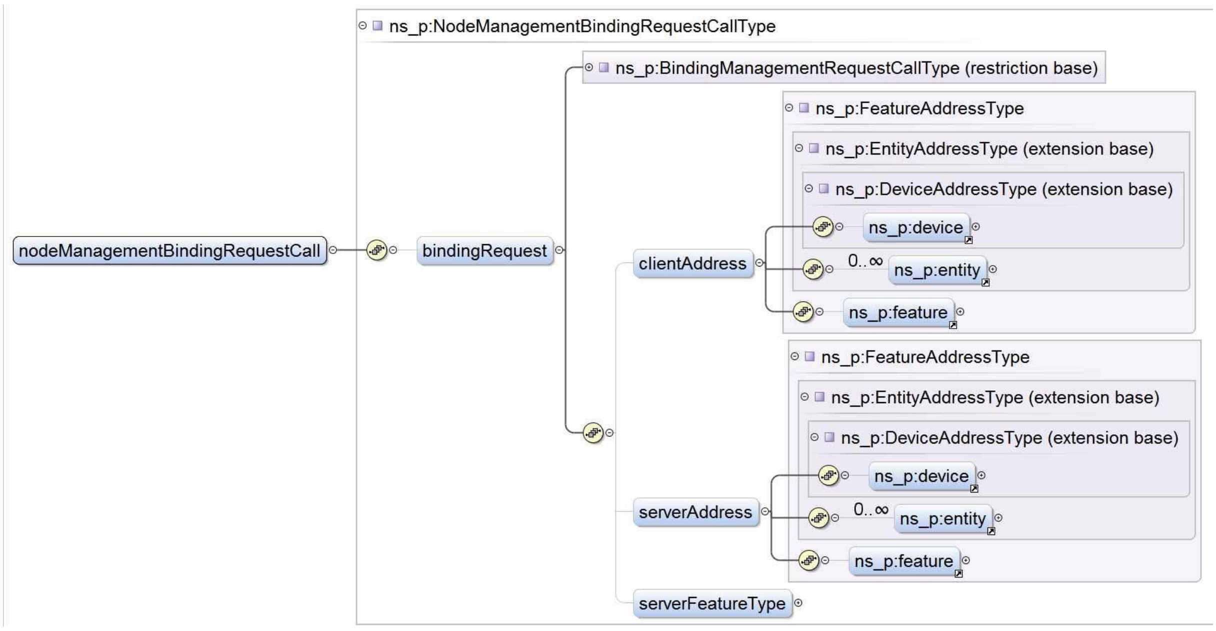


Figure 19: nodeManagementBindingRequestCall function overview

Element name	Type	M/O/NV/C (see 2.1.1)	Explanation
bindingRequest		M	SHALL be present.

bindingRequest. clientAddress		M	SHALL be present.
bindingRequest. clientAddress. device	AddressDeviceType; see "Device address" in section 7.1.1.2	O	SHOULD be present to specify the device address of the device that holds the client feature. If absent, the receiver has to identify the device via some other method.
bindingRequest. clientAddress. entity (list)	AddressEntityType (see section 2.2)	1..unbounded	The entity part(s) of the client's feature that shall be bound.
bindingRequest. clientAddress. feature	AddressFeatureType (see section 2.2)	M	SHALL state the feature of the client that shall be bound.
bindingRequest. serverAddress		M	SHALL be present.
bindingRequest. serverAddress. device	AddressDeviceType; see "Device address" in section 7.1.1.2	O	SHOULD be present to specify the device address of the device that holds the server feature. If absent, the receiver has to identify the device via some other method.
bindingRequest. serverAddress. entity (list)	AddressEntityType (see section 2.2)	1..unbounded	The entity part(s) of the server's feature that shall be bound.
bindingRequest. serverAddress. feature	AddressFeatureType (see section 2.2)	M	SHALL state the feature of the server that shall be bound.
bindingRequest. serverFeatureType	FeatureTypeType (see section 2.2)	O	SHOULD be present and state the featureType of the server's feature. A binding destination MAY consider the featureType before accepting a binding.

Table 14: Binding request with nodeManagementBindingRequestCall

The element "serverFeatureType" SHOULD be evaluated by the server to prevent unintended/unspecified bindings.

A feature server SHOULD always work according to its announced functionality, regardless of the binding partner.

Please note within certain feature types, additional rules have been specified regarding binding.

Remark: The binding request SHOULD be sent with the indication for "acknowledgement message is required" (see section 5.2.5.1). In this case the feature server EITHER responds with a "positive acknowledgement" (i.e. "application success") if it accepts the binding request OR it responds with a "negative acknowledgement" (i.e. "application error") (with "errorNumber" set to 7) if it declines the binding request.

7.3.3 Reading binding-information

In general, binding-information is organized in binding entries. The primary NodeManagement instance of a SPINE device keeps the binding entries of all its bindings (though the exchanged information is usually just a subset as it is always tailored to the communication partner as described below). Each binding entry contains information on the relation of an own feature to one feature of a binding partner. Consequently, a binding entry is specific to a binding partner.

Within this section, between two devices A and B only those binding entries are exchanged that concern the devices A and B. I.e. no binding entries of a third device C are exchanged between A and B.

The request for another node’s binding entries MAY originate from any source address. However, it is recommended that they originate from the primary NodeManagement instance.

The request for another node’s binding entries SHALL be submitted to the recipient’s primary NodeManagement instance.

The request for another node’s binding entries SHALL be sent using a “read” classifier. The content of the payload SHALL be an “empty” function “nodeManagementBindingData”, i.e. it SHALL NOT have any child element.

If the received request is valid the recipient SHALL create a response according to the usual rules (e.g. set the classifier to “reply”, set message number elements (msgCounter, msgCounterReference) accordingly, etc.). The content of payload SHALL conform to the function “nodeManagementBindingData” as described in the following table.

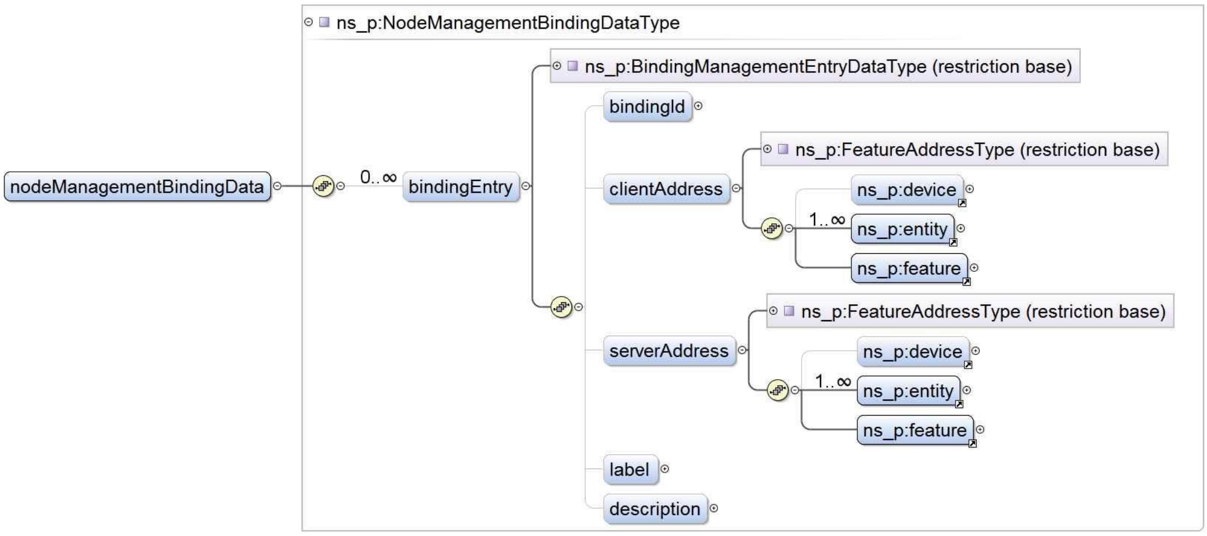


Figure 20: nodeManagementBindingData function overview

The following rules on "device" address parts in the reply function "nodeManagementBindingData" permit to reduce the size of the exchanged message to some extent. These rules also apply if the function is used in a notification:

1. Absence of BOTH "bindingEntry. clientAddress. device" AND "bindingEntry. serverAddress. device":
   This combination is only valid if the respective bindingEntry instance denotes a "feature server" of the originator of this reply or notify function instance:
   a. The absence of "bindingEntry. clientAddress. device" SHALL be treated as if it was present and set to the recipient's "device" address part.

    b. The absence of "bindingEntry. serverAddress. device" SHALL be treated as if it was present and set to the sender's "device" address part.

2. Absence of EITHER "bindingEntry. clientAddress. device" OR "bindingEntry. serverAddress. device":
   This combination is only valid to omit the "device" address part of the recipient of this reply or notify function instance. I.e. the sender's "device" address part SHALL be present in the respective element for this combination. This means the absent element SHALL be treated as if it was present and set to the recipient's "device" address part: Considering a specific binding entry with exactly one "device" address part, if the entry contains
    a. a server feature of the sender of this reply or notify function instance: the "device" address part of "serverAddress" is present and set to the sender's device address;
    b. a client feature of the sender of this reply or notify function instance: the "device" address part of "clientAddress" is present and set to the sender's device address.

Example: We assume a client feature of device "A" has a binding to a server feature of device "B". If "A" asks for "B"'s binding information, then "B" can send a reply where the "device" address part in "clientAddress" of this binding entry is absent and the "device" address part in "serverAddress" is set to "B"'s device address (in this case item 1 above is as well possible). On the other hand, if "B" asks for the binding information of "A", the response of "A" must have the "device" address part in "clientAddress" set to the device address of "A", but it can leave out the "device" address part in "serverAddress".

Please note that these items need to be distinguished carefully as the rules can lead to different results. Please also note that these rules do NOT specify whether/which "device" information needs to be stored - these rules just permit to make a message smaller.

Element name	**Type**	**M/O/NV/C (see 2.1.1)**	**Explanation**
bindingEntry (list)		0..unbounded	SHALL be present if binding entries are available for the recipient. Otherwise, it SHALL not be present.
bindingEntry. bindingId	xs:unsignedInt	O	MAY be present. If present it SHALL contain a server's unique ID of the binding for the corresponding binding entry.
bindingEntry. clientAddress		M	SHALL be present.
bindingEntry. clientAddress. device	AddressDeviceType; see "Device address" in section 7.1.1.2	O	SHOULD be present to specify the device address of the device that holds the client feature. If absent, the rules described in the text apply.
bindingEntry. clientAddress. entity (list)	AddressEntityType (see section 2.2)	1..unbounded	SHALL be present to specify the entity address part(s) of the client.
bindingEntry. clientAddress. feature	AddressFeatureType (see section 2.2)	M	SHALL be present to specify the feature address of the client.
bindingEntry. serverAddress		M	SHALL be present.

bindingEntry. serverAddress. device	AddressDeviceType; see “Device address” in section 7.1.1.2	O	SHOULD be present to specify the device address of the device that holds the server feature. If absent, the rules described in the text apply.
bindingEntry. serverAddress. entity (list)	AddressEntityType (see section 2.2)	1..unbounded	SHALL be present to specify the entity address part(s) of the server.
bindingEntry. serverAddress. feature	AddressFeatureType (see section 2.2)	M	SHALL be present to specify the feature address of the server.
bindingEntry. label	*Common data type "LabelType". See section 2.2.*	O	MAY be present and store additional short information about this binding.
bindingEntry. description	xs:string	O	MAY be present and store additional information about this binding. The string-length SHOULD NOT be longer than 256 characters. If it is longer, the sender SHALL consider the possibility that the receiver will shorten the string to 256 characters.

Table 15: nodeManagementBindingData holds list of binding entries

7.3.4 Release of a binding

This section discusses how an existing binding between two nodes A and B can be released. It does NOT discuss how a third node C can release a binding between the nodes A and B.

Either binding partner can request the deletion of a binding. Releasing a binding between the devices A and B also includes the deletion of the corresponding binding entry of the node A as well as the one of node B. Subsequently we assume device A initiates the request to release the binding.

All "delete" operations respect the "role-relation". I.e. address parts of "clientAddress" apply ONLY to "feature clients" and address parts of "serverAddress" apply ONLY to "feature servers".

All "delete" operations SHALL use the function “nodeManagementBindingDeleteCall” with a "call" classifier:

In order to delete a binding between a specific client feature address of device A and a specific server feature address of device B the clientAddress and serverAddress SHALL be set properly in the function “nodeManagementBindingDeleteCall”.

In order to delete all bindings of the same "role-relation" between a specific entity of device A and a specific entity of device B the “entity” elements in the function “nodeManagementBindingDeleteCall” SHALL be set to the specific values and the “feature” values in this function SHALL NOT be present.

In order to delete all bindings of the same "role-relation" between the devices (i.e. independent from any specific entity or feature value) the “entity” elements and the “feature” elements in the function “nodeManagementBindingDeleteCall” SHALL NOT be present.

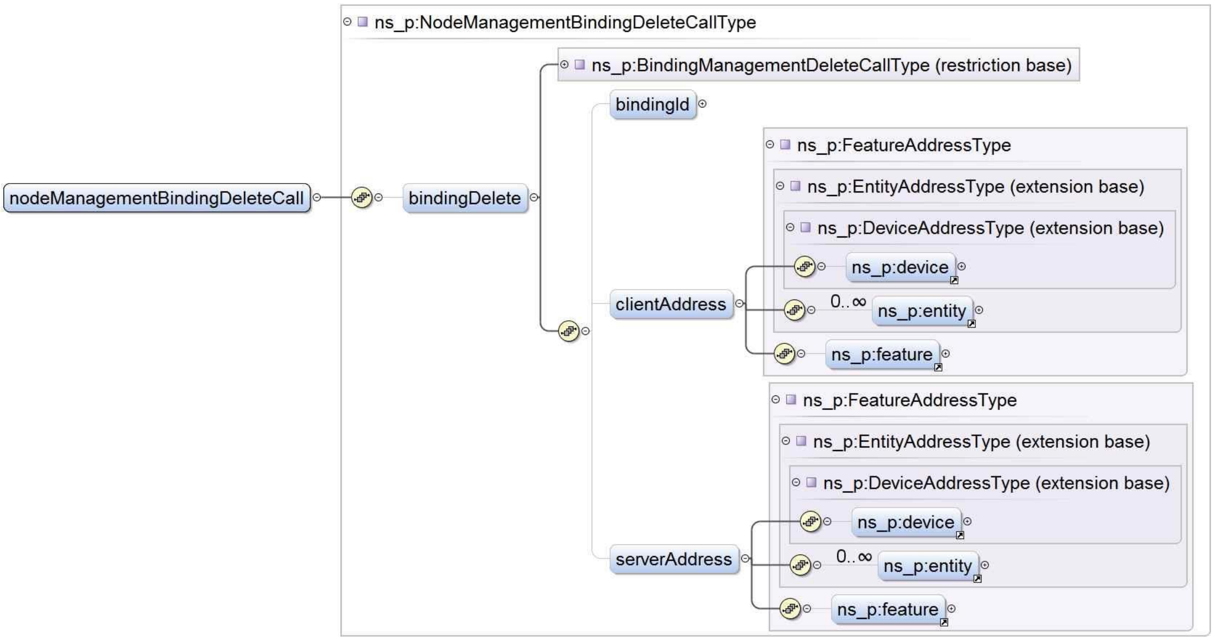


Figure 21: nodeManagementBindingDeleteCall function overview

The following rules on "device" address parts in the function "nodeManagementBindingDeleteCall" permit to reduce the size of the exchanged message to some extent:

1. Absence of BOTH "bindingDelete. clientAddress. device" AND "bindingDelete. serverAddress. device":
   This combination is only valid if the respective bindingDelete instance denotes a "feature client" (or potentially several feature clients if "feature" address parts or even "entity" address parts are omitted) of the originator of this function instance (i.e. this is the "delete" counterpart to the client's binding request):
   a. The absence of "bindingDelete. clientAddress. device" SHALL be treated as if it was present and set to the sender's "device" address part.
   b. The absence of "bindingDelete. serverAddress. device" SHALL be treated as if it was present and set to the recipient's "device" address part.
2. Absence of EITHER "bindingDelete. clientAddress. device" OR "bindingDelete. serverAddress. device":
   This combination is only valid to omit the "device" address part of the recipient of this function. I.e. the sender's "device" address part SHALL be present in the respective element for this combination. This means the absent element SHALL be treated as if it was present and set to the recipient's "device" address part: Considering a specific binding entry with exactly one "device" address part, if the entry contains
   a. a server feature of the sender of this deletion request: the "device" address part of "serverAddress" is present and set to the sender's device address;
   b. a client feature of the sender of this deletion request: the "device" address part of "clientAddress" is present and set to the sender's device address.

Please note that these items need to be distinguished carefully as the rules can lead to different results.

The rules on omitting "device" address parts are compatible with the specified deletion requests where "feature" or even "entity" address parts are omitted.

Element name	Type	M/O/NV/C (see 2.1.1)	Explanation
bindingDelete		M	SHALL be present.
bindingDelete. bindingId	xs:unsignedInt	O	SHOULD NOT be present. If present, the value SHALL be ignored.
bindingDelete. clientAddress		M	SHALL be present.
bindingDelete. clientAddress. device	AddressDeviceType; see “Device address” in section 7.1.1.2	O	SHOULD be present to specify the device address of the device that holds the client feature. If absent, the rules described in the text apply.
bindingDelete. clientAddress. entity (list)	AddressEntityType (see section 2.2)	0..unbounded See right, (*1).	If used, SHALL state the client's entity address part(s) of this binding. (*1) Element presence: If a specific binding shall be deleted: M If all bindings of an entity (including its sub-entities) shall be deleted: M If all bindings of a device shall be deleted: NV
bindingDelete. clientAddress. feature	AddressFeatureType (see section 2.2)	See right, (*1).	If used, SHALL state the client's feature address of this binding. (*1) Element presence: If a specific binding shall be deleted: M If all bindings of an entity shall be deleted: NV If all bindings of a device shall be deleted: NV
bindingDelete. serverAddress		M	SHALL be present.
bindingDelete. serverAddress. device	AddressDeviceType; see “Device address” in section 7.1.1.2	O	SHOULD be present to specify the device address of the device that holds the server feature. If absent, the rules described in the text apply.
bindingDelete. serverAddress. entity (list)	AddressEntityType (see section 2.2)	0..unbounded See right, (*1).	If used, SHALL state the server's entity address part(s) of this binding. (*1) Element presence: If a specific binding shall be deleted: M If all bindings of an entity shall be deleted: M If all bindings of a device shall be deleted: NV

bindingDelete. serverAddress. feature	AddressFeatureType (see section 2.2)	See right, (*1).	If used, SHALL state the server's feature address of this binding. (*1) Element presence: If a specific binding shall be deleted: M If all bindings of an entity (including its sub-entities) shall be deleted: NV If all bindings of a device shall be deleted: NV

Table 16: Remove Binding with nodeManagementBindingDeleteCall

Remark: The request to release a binding SHOULD be sent with the indication for “acknowledgement message is required” (see section 5.2.5.1).

7.3.5 Renew lost binding

Section 7.3.6 describes situations where binding information may get lost for some reason. Esp. in case a feature server loses binding information without prior notice by a feature client, the client needs a possibility to renew a binding entry if necessary. First of all, this requires a proper detection of a lost binding:

If a server receives a command (with acknowledgement indication) where the feature server requires that it originates from a binding partner, but the server has no corresponding binding entry for the requesting feature client stored, the server SHALL reply with a "negative acknowledgement" with "errorNumber" set to 9.

If a client receives this kind of negative acknowledgement it can then decide whether it creates a binding (again) or not.

Please note that even a request for a "renewed" binding may be declined by the feature server. This can esp. happen if the feature server just permits a single binding and had been configured with a binding to a different feature client in between.

7.3.6 Considerations on broken bindings (informative)

SPINE nodes store bindings and subscriptions persistently and deliver events to their binding and subscription partners. However, the binding and subscription partners might be unreachable for a rather short or long time.

The reasons for temporarily or permanently unreachable binding or subscription partners might be various. Some SPINE nodes might not be reachable for different periods of time, because someone temporarily unplugged their main power source. Some other SPINE nodes might not be reachable because they have been permanently removed from the network, because the user has sold the device or because the device has been permanently damaged and replaced.

A device cannot automatically and accurately determine why its binding or subscriptions partners are currently unreachable. This can lead to an increasing number of “dead bindings” or “dead subscriptions". This is why manufacturers need to provide a (proprietary) mechanism to recover from such dead bindings (see rules in section 7.3.1). One possible solution is to provide a factory reset.

Although subscriptions and bindings are stored persistently, there might be situations where a SPINE node forgets its subscriptions and bindings (e.g. if factory reset is used while other subscription partners are not reachable and therefore the bindings and subscriptions are not released properly). This can be another cause of “dead bindings” or “dead subscriptions” that are only active on one side but forgotten by the other side.

However, it is recommended to check and possibly remove or renew bindings/subscriptions (see section 7.3.5 renew bindings and section 7.4.5 renew subscriptions) in any of the following cases:

a) a device assumes that no more notifications are received from a subscription partner
b) notifications are not acknowledged by other device for long period
c) a device assumes that no more writes are received from a binding partner
d) writes are not acknowledged by the other device for long period
e) the own device was offline
f) result message with errorNumber 9 or other error was received

It is considered as error if:

a) unexpected notifications are received from a device not known as subscription partner
b) unexpected writes are received from a device not know as binding partner

A device MAY also remove bindings/subscriptions if a device assumes that the binding or subscription partner is inactive over a long period of time, e.g. no more acknowledges are received for writes or notifications over a long period of time.

7.4 Subscription

7.4.1 Basic definitions and rules

As currently only the subscription on a whole feature is possible the term subscription always relates to a full feature subscription within this specification. This means also if only a certain part of the feature might be relevant for a subscriber, e.g. only a certain function part of a feature, with full feature subscription a subscriber might also receive notification of other feature parts that might not be relevant for the subscriber. In this case the subscriber may ignore those other feature parts.

Subscription is used by a feature client to tell a feature server that the feature client wants to be notified about changes of the feature server's data. For this purpose, feature clients can subscribe to feature servers and will then be notified about data changes (please note that a subscribed client is as well notified if it caused the change by a "write" operation; this is because "write operations" (with or without acknowledgements) and subscriptions are independent mechanisms).

Subscription SHALL be supported by a SPINE node

1. if it has at least one feature server that REQUIRES subscription for specific tasks, or
2. if it has at least one feature client that needs to use a feature server that REQUIRES subscription for specific tasks.

Please note that some feature types may define requirements for subscription! The "subscription" concept specific functions SHALL be implemented on the primary NodeManagement instance.

Establishing a subscription by a feature client requires knowledge of the communication partner's server features. Thus, it is recommended to get this information from a detailed discovery procedure as described in section 7.1. Of course, if during detailed discovery no server features to subscribe to have been found, no subscription will be established.

A subscription is a special functional relation between one feature with the role “server” (called “feature server”) and one feature with the role “client” (called “feature client”). In addition, a feature with a role "special" can be considered as "feature client" or "feature server" (or both) as described below and can then be used for a subscription accordingly. In general, subscriptions have a lower priority than bindings. Similar to bindings, subscriptions usually include clear responsibilities and permissions between the “feature server” and the “feature client(s)”. But in contrast to bindings the rules for subscriptions are simpler. The following rules hold for every subscription between features with standardized featureTypes:

1. Notification: A “feature server” MAY notify data to a “feature client”. A “feature client” SHALL NOT notify data to a “feature server”.
2. Regarding the establishment of a subscription the following rules can be described: The request to establish a subscription SHALL originate from the node with the “feature client”. It SHALL NOT originate from the node with the “feature server”.
3. The node with the “feature server” MAY deny a subscription request.
   Remark: This could happen if the server already has a subscription with another node and no further subscription is possible, or if the trust level requirements are not fulfilled.
4. Subscription information SHALL be kept persistently by both subscription partners unless the subscription shall be released explicitly.
5. It is not possible to create a subscription to entities or devices. It is only possible to create a subscription to a feature.
6. Either subscription partner is permitted to release the subscription.
7. An implementation SHALL enable a user to release a subscription (i.e. to remove it) on a node even if the subscription partner is not available anymore.
8. Requests to establish or remove a subscription can be performed at any time.
9. In general, the subscription concept also permits to model subscription relations between features of the same device (i.e. device-internal subscriptions). In this case, the rules on absent "device" address parts (see note at the end) SHALL NOT be used to distinguish between "feature client" and "feature server". Rather, the roles of the features need to be used and considered explicitly in order to model a unique relation. Please note that this is not possible if two different features of the same device both have the role "special".
   Note on rules on absent "device" address parts: The following sections specify some functions with child elments "... clientAddress. device" and "... serverAddress. device" (see Table 17, Table 18, Table 19, e.g.; the parent elements are abbreviated here with "..."). These elements can contain device address parts. The explanations in the sections or function definitions contain rules on the absence of these elements.

A feature with the role "special" may have own data or it may consume other device's data (or both). If a "special" feature sends own data (i.e. send reply or notify) or receives a "write" operation to change its own data it acts as "feature server". If a "special" feature receives data owned by another device (i.e. receive reply or notify) or sends a "write" operation to change data it acts as "feature client".

The major benefit of a subscription is to arrange notification of changes. As an example, a subscription is established between a room temperature control system acting as “Measurement” client and a room temperature sensor acting as “Measurement” server. Once a subscription on sensor's "measurement" server feature is established, the sensor knows where to send the measured room temperature to and the control system knows which sensor needs to be considered. Moreover, the sensor knows it is responsible to provide the control system with new values in case of (relevant) temperature changes.

7.4.2 Create Subscription

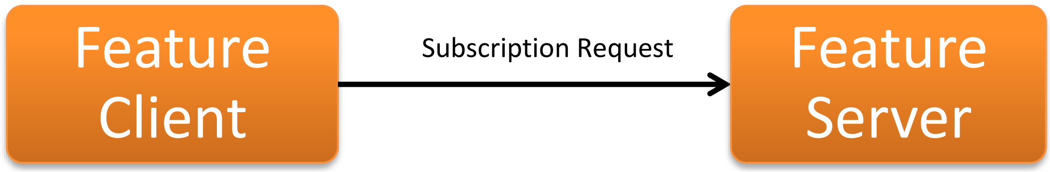


Figure 22: Subscription request

The nodeManagementSubscriptionRequestCall SHALL always be sent using a “call” classifier. The content of payload is described in the following table.

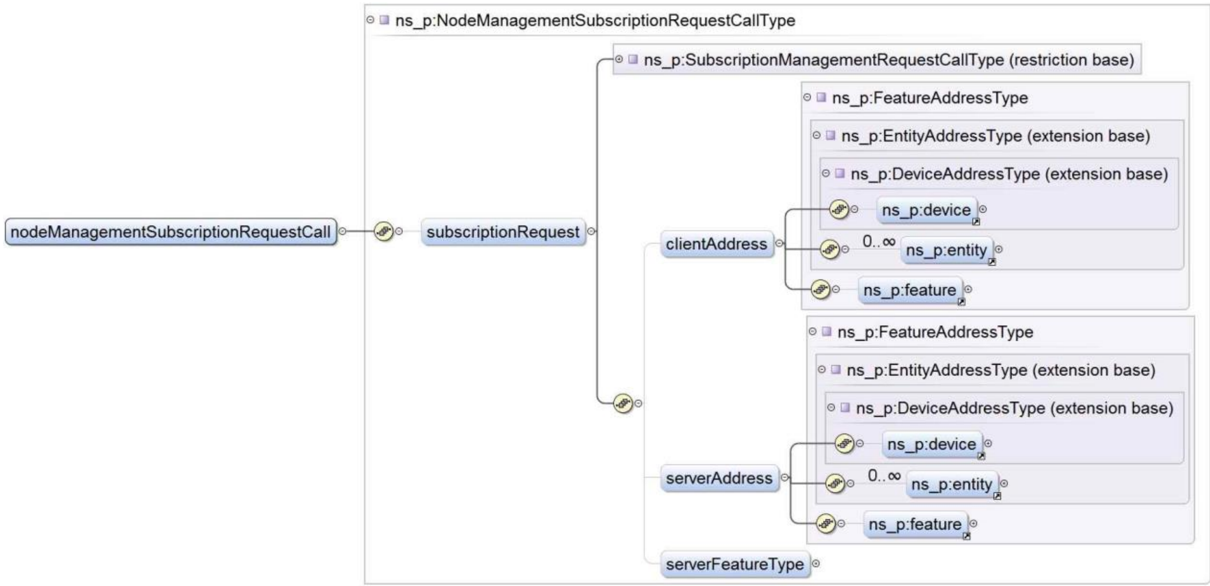


Figure 23: nodeManagementSubscriptionRequestCall function overview

Element name	Type	M/O/NV/C (see 2.1.1)	Explanation
subscriptionRequest		M	SHALL be present.
subscriptionRequest. clientAddress		M	SHALL be present.

subscriptionRequest. clientAddress. device	AddressDeviceType; see “Device address” in section 7.1.1.2	O	SHOULD be present to specify the device address of the device that holds the client feature. If absent, the receiver has to identify the device via some other method.
subscriptionRequest. clientAddress. entity (list)	AddressEntityType (see section 2.2)	1..unbounde d	The entity part(s) of the client's feature that requests the subscription.
subscriptionRequest. clientAddress. feature	AddressFeatureType (see section 2.2)	M	SHALL state the feature of the client that requests the subscription
subscriptionRequest. serverAddress		M	SHALL be present.
subscriptionRequest. serverAddress. device	AddressDeviceType; see “Device address” in section 7.1.1.2	O	SHOULD be present to specify the device address of the device that holds the server feature. If absent, the receiver has to identify the device via some other method.
subscriptionRequest. serverAddress. entity (list)	AddressEntityType (see section 2.2)	1..unbounde d	The entity part(s) of the server's feature that is requested for subscription.
subscriptionRequest. serverAddress. feature	AddressFeatureType (see section 2.2)	M	SHALL state the feature of the server that is requested for subscription.
subscriptionRequest. serverFeatureType	FeatureTypeType (see section 2.2)	O	SHOULD be present and state the featureType of the server's feature.

Table 17: Subscription request with nodeManagementSubscriptionRequestCall

A SPINE node SHALL evaluate whether a call for establishing a subscription suits the trust level of the according calling SPINE node. Furthermore, the sender of the subscription request SHALL NOT expect that the feature server adjusts its feature’s functionality according to the subscription request. To put it in other words: A feature server SHOULD always work according to its announced functionality, regardless of the subscription partner.

Remark: The subscription request SHOULD be sent with the indication for “acknowledgement message is required” (see section 5.2.5.1).

7.4.3 Reading subscription information

In general, subscription information is organized in subscription entries. The primary NodeManagement instance keeps the subscription entries of all entities and features (though the exchanged information is usually just a subset as it is always tailored to the communication partner as described below). Each subscription entry contains information on the relation of an own feature to one feature of a subscription partner. Consequently, a subscription entry is specific to a communication partner.

Within this section, between two devices A and B only those subscription entries are exchanged that concern the devices A and B. I.e. no subscription entries of a third device C are exchanged between A and B.

The request for another node’s subscription entries MAY originate from any source address. However, it is recommended to originate from the primary NodeManagement instance.

The request for another node’s subscription entries SHALL be submitted to the recipient’s primary NodeManagement instance.

The request for another node’s subscription entries SHALL be sent using a “read” classifier. The content of payload SHALL be an “empty” function “nodeManagementSubscriptionData”, i.e. it SHALL NOT have any child element.

If the received request is valid the recipient SHALL create a response according to the usual rules (e.g. set the classifier to “reply”, set message number elements (msgCounter, msgCounterReference) accordingly, etc.). The content of payload SHALL be the “nodeManagementSubscriptionData” function.

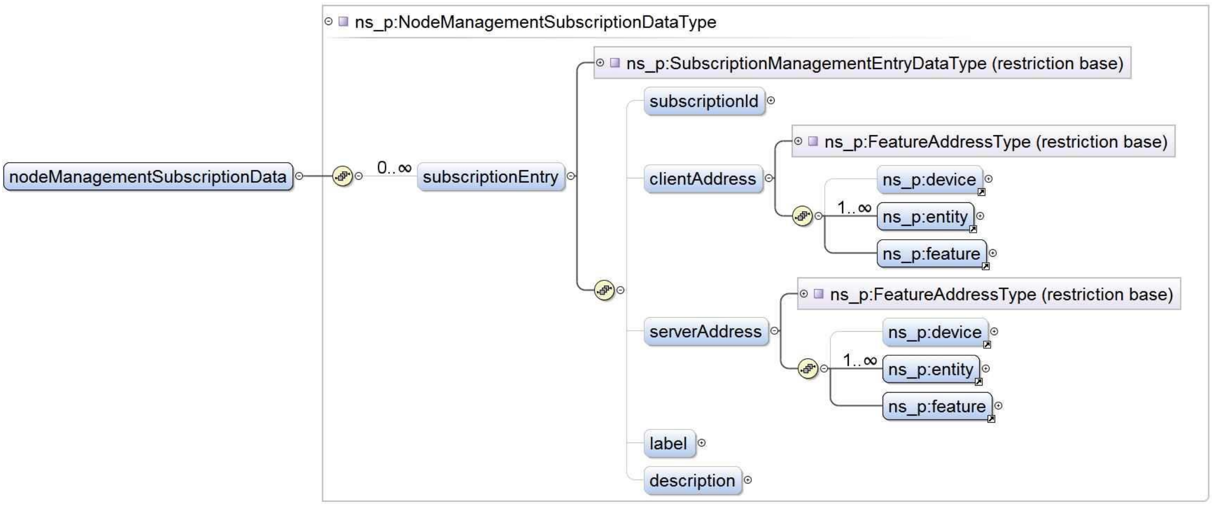


Figure 24: nodeManagementSubscriptionData function overview

The following rules on "device" address parts in the reply function "nodeManagementSubscriptionData" permit to reduce the size of the exchanged message to some extent. These rules also apply if the function is used in a notification:

1. Absence of BOTH "subscriptionEntry. clientAddress. device" AND "subscriptionEntry. serverAddress. device":
   This combination is only valid if the respective subscriptionEntry instance denotes a "feature server" of the originator of this reply or notify function instance:
   a. The absence of "subscriptionEntry. clientAddress. device" SHALL be treated as if it was present and set to the recipient's "device" address part.
   b. The absence of "subscriptionEntry. serverAddress. device" SHALL be treated as if it was present and set to the sender's "device" address part.
2. Absence of EITHER "subscriptionEntry. clientAddress. device" OR "subscriptionEntry. serverAddress. device":
   This combination is only valid to omit the "device" address part of the recipient of this reply or

notify function instance. I.e. the sender's "device" address part SHALL be present in the respective element for this combination. This means the absent element SHALL be treated as if it was present and set to the recipient's "device" address part: Considering a specific subscription entry with exactly one "device" address part, if the entry contains

a. a server feature of the sender of this reply or notify function instance: the "device" address part of "serverAddress" is present and set to the sender's device address;
b. a client feature of the sender of this reply or notify function instance: the "device" address part of "clientAddress" is present and set to the sender's device address.

Example: We assume a client feature of device "A" has a subscription to a server feature of device "B". If "A" asks for "B"'s subscription information, then "B" can send a reply where the "device" address part in "clientAddress" of this subscription entry is absent and the "device" address part in "serverAddress" is set to "B"'s device address (in this case item 1 above is as well possible). On the other hand, if "B" asks for the subscription information of "A", the response of "A" must have the "device" address part in "clientAddress" set to the device address of "A", but it can leave out the "device" address part in "serverAddress".

Please note that these items need to be distinguished carefully as the rules can lead to different results. Please also note that these rules do NOT specify whether/which "device" information needs to be stored - these rules just permit to make a message smaller.

Element name	**Type**	**M/O/NV/C (see 2.1.1)**	**Explanation**
subscriptionEntry (list)		0..unbounded	SHALL be present if subscription entries are available for the recipient. Otherwise, it SHALL not be present.
subscriptionEntry. subscriptionId	xs:unsignedInt	O	MAY be present. If present it SHALL contain a server's unique ID of the subscription.
subscriptionEntry. clientAddress		M	SHALL be present.
subscriptionEntry. clientAddress. device	AddressDeviceType; see "Device address" in section 7.1.1.2	O	SHOULD be present to specify the device address of the device that holds the client feature. If absent, the rules described in the text apply.
subscriptionEntry. clientAddress. entity (list)	AddressEntityType (see section 2.2)	1..unbounded	SHALL be present to specify the entity address part(s) of the client.
subscriptionEntry. clientAddress. feature	AddressFeatureType (see section 2.2)	M	SHALL be present to specify the feature address of the client.
subscriptionEntry. serverAddress		M	SHALL be present.
subscriptionEntry. serverAddress. device	AddressDeviceType; see "Device address" in section 7.1.1.2	O	SHOULD be present to specify the device address of the device that holds the server feature. If absent, the rules described in the text apply.

subscriptionEntry. serverAddress. entity (list)	AddressEntityType (see section 2.2)	1..unbounded	SHALL be present to specify the entity address part(s) of the server.
subscriptionEntry. serverAddress. feature	AddressFeatureType (see section 2.2)	M	SHALL be present to specify the feature address of the server.
subscriptionEntry. label	*Common data type "LabelType". See section 2.2.*	O	MAY be present and store additional information about this subscription. The string-length SHOULD NOT be longer than 256 characters. If it is longer, the sender SHALL consider the possibility that the receiver will shorten the string to 256 characters.
subscriptionEntry. description	*Common data type "DescriptionType". See section 2.2.*	O	MAY be present and store additional information about this subscription. The string-length SHOULD NOT be longer than 4096 characters. If it is longer, the sender SHALL consider the possibility that the receiver will shorten the string to 4096 characters.

Table 18: nodeManagementSubscriptionData holds list of subscription entries

7.4.4 Release of a subscription

This section discusses how an existing subscription between two devices A and B can be released. It does NOT discuss how a third device C can release a subscription between the devices A and B.

Either subscription partner can request for the deletion of a subscription. Releasing a subscription between the devices A and B also includes the deletion of the corresponding subscription entry of the device A as well as the one of device B. Subsequently we assume device A initiates the request to release the subscription.

All "delete" operations respect the "role-relation". I.e. address parts of "clientAddress" apply ONLY to "feature clients" and address parts of "serverAddress" apply ONLY to "feature servers".

All "delete" operations SHALL use the function “nodeManagementSubscriptionDeleteCall” with a "call" classifier:

In order to delete a subscription between a specific client feature address of device A and a specific server feature address of device B the clientAddress and serverAddress SHALL be set properly in the function “nodeManagementSubscriptionDeleteCall”.

In order to delete all subscriptions of the same "role-relation" between a specific entity of device A and a specific entity of device B the “entity” elements in the function “nodeManagementSubscriptionDeleteCall” SHALL be set to the specific values and the “feature” values in this function SHALL NOT be present.

In order to delete all subscriptions of the same "role-relation" between the devices (i.e. independent from any specific entity or feature value) the “entity” elements and the “feature” elements in the function “nodeManagementSubscriptionDeleteCall” SHALL NOT be present.

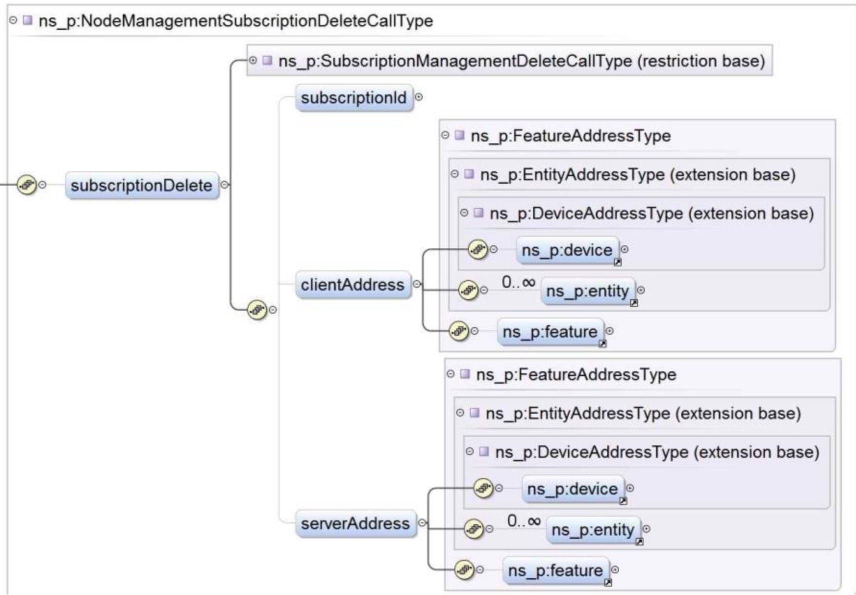


Figure 25: nodeManagementSubscriptionDeleteCall function overview

The following rules on "device" address parts in the function "nodeManagementSubscriptionDeleteCall" permit to reduce the size of the exchanged message to some extent:

1. Absence of BOTH "subscriptionDelete. clientAddress. device" AND "subscriptionDelete. serverAddress. device":
   This combination is only valid if the respective subscriptionDelete instance denotes a "feature client" (or potentially several feature clients if "feature" address parts or even "entity" address parts are omitted) of the originator of this function instance (i.e. this is the "delete" counterpart to the client's subscription request):
   a. The absence of "subscriptionDelete. clientAddress. device" SHALL be treated as if it was present and set to the sender's "device" address part.
   b. The absence of "subscriptionDelete. serverAddress. device" SHALL be treated as if it was present and set to the recipient's "device" address part.
2. Absence of EITHER "subscriptionDelete. clientAddress. device" OR "subscriptionDelete. serverAddress. device":
   This combination is only valid to omit the "device" address part of the recipient of this function. I.e. the sender's "device" address part SHALL be present in the respective element for this combination. This means the absent element SHALL be treated as if it was present and set to the recipient's "device" address part: Considering a specific subscription entry with exactly one "device" address part, if the entry contains
   a. a server feature of the sender of this deletion request: the "device" address part of "serverAddress" is present and set to the sender's device address;
   b. a client feature of the sender of this deletion request: the "device" address part of "clientAddress" is present and set to the sender's device address.

Please note that these items need to be distinguished carefully as the rules can lead to different results.

The rules on omitting "device" address parts are compatible with the specified deletion requests where "feature" or even "entity" address parts are omitted.

Element name	Type	M/O/NV/C (see 2.1.1)	Explanation
subscriptionDelete		M	SHALL be present.
subscriptionDelete. subscriptionId	xs:unsignedInt	O	SHOULD NOT be present. If present, the value SHALL be ignored.
subscriptionDelete. clientAddress		M	SHALL be present.
subscriptionDelete. clientAddress. device	AddressDeviceType; see “Device address” in section 7.1.1.2	O	SHOULD be present to specify the device address of the device that holds the client feature. If absent, the rules described in the text apply.
subscriptionDelete. clientAddress. entity (list)	AddressEntityType (see section 2.2)	0..unbounded See right, (*1).	If used, SHALL state the client's entity address part(s) of this subscription. (*1) Element presence: If a specific subscription shall be deleted: M If all subscriptions of an entity (including its sub-entities) shall be deleted: M If all subscription of a device shall be deleted: NV
subscriptionDelete. clientAddress. feature	AddressFeatureType (see section 2.2)	See right, (*1).	If used, SHALL state the client's feature address of this subscription. (*1) Element presence: If a specific subscription shall be deleted: M If all subscription of an entity (including its sub-entities) shall be deleted: NV If all subscription of a device shall be deleted: NV
subscriptionDelete. serverAddress		M	SHALL be present.
subscriptionDelete. serverAddress. device	AddressDeviceType; see “Device address” in section 7.1.1.2	O	SHOULD be present to specify the device address of the device that holds the server feature. If absent, the rules described in the text apply.
subscriptionDelete. serverAddress. entity (list)	AddressEntityType (see section 2.2)	0..unbounded See right, (*1).	If used, SHALL state the server's entity address part(s) of this subscription. (*1) Element presence: If a specific subscription shall be deleted: M If all subscription of an entity (including its sub-entities) shall be deleted: M

			If all subscription of a device shall be deleted: NV
subscriptionDelete. serverAddress. feature	AddressFeatureType (see section 2.2)	See right, (*1).	If used, SHALL state the server's feature address of this subscription. (*1) Element presence: If a specific subscription shall be deleted: M If all subscription of an entity shall be deleted: NV If all subscription of a device shall be deleted: NV

Table 19: Remove subscription with nodeManagementSubscriptionDeleteCall

Remark: The request to release a subscription SHOULD be sent with the indication for “acknowledgement message is required” (see section 5.2.5.1).

7.4.5 Renewal of subscription

Section 7.4.6 describes situations where subscription information may get lost for some reason. It is also possible that a given feature is not always available, hence any subscription to it needs to be treated dynamically. Both cases may require a renewal of a subscription:

a) If a client suspects that a subscription is not valid anymore (e.g. the server lost or removed the subscription) and it still needs the subscription: In this case the client SHOULD renew the subscription by creating a proper subscription as described in section 7.4.2.
b) If a server or client wants to remove a feature with subscriptions: The server or client should first release subscriptions for the corresponding feature as described in section 7.4.4. Then it can remove its feature. If the corresponding feature reappears the client can subscribe again as described in section 7.4.2.

Please note that a feature server may decline subscription creation, even if a subscription is just renewed from the client's point of view. This can esp. happen if the feature server just permits a single subscription and had been configured with a subscription to a different feature client in between or if the old subscription for this client was deleted deliberately by the server (i.e. if the server intentionally blocks subscription requests of this specific client).

7.4.6 Considerations on broken subscriptions (informative)

See section 7.3.6 for further information.

7.5 Use Case discovery

7.5.1 Basic definitions and rules

The use case discovery allows to discover which use cases are supported and which actor a device embodies in a corresponding use case. This allows to derive information about which data a device supports as a client or as a server, as defined by each use case scenario.

In general, if all data needed for a use case is present at a server, the use case is possible with this server, no matter if the use case discovery includes the use case or not.

However, the use case discovery is the only way to discover which data is supported by a client, while the server data can be discovered with detailed discovery and reads on the corresponding features. However, the detailed discovery and feature evaluation on a server only shows data that is currently available, but in certain cases the necessary data might not always be available. This means that the detailed discovery may show less data than derivable from the use case discovery. E.g. data of a flexible forecast is only offered by a washing machine if the user has configured the washing machine accordingly. In this case the use case discovery is the only possibility to discover the support of flexible forecast in advance. This means the mismatch between use case discovery and the content of corresponding server features allows to derive which use cases are currently available on the server interface and which use cases are currently not available but may be available at a later point in time.

The detailed discovery may also show more data than derivable from the use case discovery. This can have several reasons, e.g. as a device is not obliged to allow discovery of all supported use cases or a device has implemented use cases not yet officially added within the SPINE use case discovery.

The use case discovery is optional in general. However, if the nodeManagementUseCaseData function is supported within nodeManagement, the following rules apply:

Official EEBUS use cases in which the device acts as client or where the required server data (e.g. Entity Type, Feature Type, function or data within a function, e.g. list entry or certain required elements) may not be available at all times SHALL be stated in the use case discovery, unless this is forbidden by device policy (e.g. trust level) or user settings (e.g. user has deactivated use case). All other use cases SHOULD be present, so the use case discovery offers a complete overview of all supported use cases.

The Use Case Discovery is part of the primary NodeManagement instance. The discovery is basically a simple read on the function nodeManagementUseCaseData which holds a list of supported Use Cases. The supported Use Cases are ordered by address and version. This means for the same address and use case version, more than one use case can be supported. This can be helpful e.g. if multiple use cases are supported on the same client feature.

The nodeManagementUseCaseData function SHALL be accessible with full read (Use Case Discovery “all at once”) and also SHOULD be accessible with partial read (Partial Use Case Discovery). Additionally, Subscription SHALL be supported, so a device is informed with a notify if there are changes regarding the support of certain use cases.

7.5.2 Use Case Discovery “all at once”

In this case a full read is performed on the nodeManagementUseCaseData function.

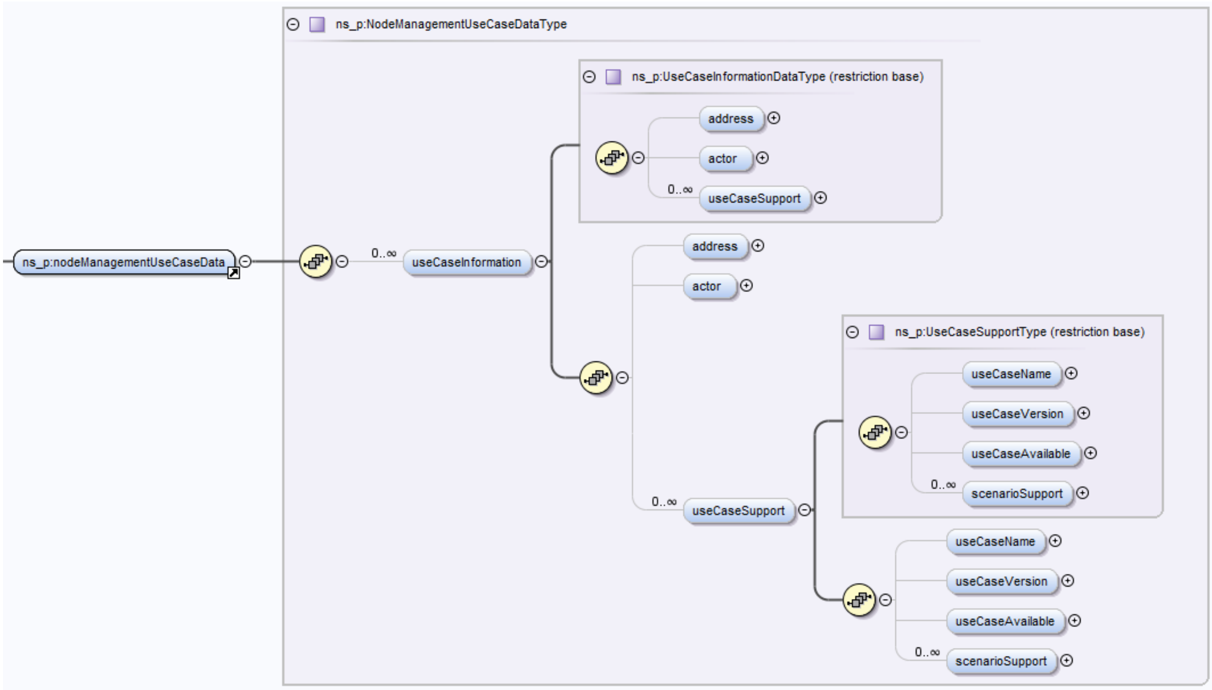


Figure 26: nodeManagementUseCaseData function

Data rules of the nodeManagementUseCaseData function:

Element name	**Type**	**M/O/NV/C**	**Explanation**
useCaseInformation (list)		O 0..unbounded	Holds information about use cases available on a specific address.
useCaseInformation. address	FeatureAddressType	M	SHALL hold a Device, Entity or Feature address. The address SHALL be a static address, which means the address SHALL be visible within detailed discovery at all times. In case of dynamic Features or Entities that MAY not be present at all times within the detailed discovery, the nearest Entity above the corresponding Feature or Entity SHALL be used as address. The whole use case functionality SHALL be accessible behind this address as soon as the dynamic or static use case functionality is available. If there is no Entity above, the device address SHALL be used.
useCaseInformation. actor	UseCaseActorType	M	Union that describes which actor is embodied. UseCaseActorEnumType is reserved for actors from official EEBUS use cases, while the

			EnumExtendType also allows to specify manufacturer specific use case actors. The string-length SHOULD NOT be longer than 128 characters. If it is longer, the sender SHALL consider the possibility that the receiver will shorten the string to 128 characters.
useCaseInformation. useCaseSupport (list)		M	Holds a list of supported use cases
useCaseInformation. useCaseSupport. useCaseName	UseCaseNameType	M	Union that describes the use case name. UseCaseNameEnumType is reserved for official EEBUS use case names, while the EnumExtendType also allows to specify manufacturer specific use case names. The string-length SHOULD NOT be longer than 128 characters. If it is longer, the sender SHALL consider the possibility that the receiver will shorten the string to 128 characters.
useCaseInformation. useCaseSupport. useCaseVersion	SpecificationVersionType	M	Holds the version number of the use case
useCaseInformation. useCaseSupport. useCaseAvailable	xs:boolean	O	If the actor also acts as client within the use case, useCaseAvailable = false SHALL be set, if the client portion of a use case is currently not available. If the client portion of the use case is available, the useCaseAvailable flag MAY be omitted. This means, if an actor has client functionality within a use case, and the useCaseAvailable flag is omitted, this SHALL be interpreted as useCaseAvailable = true. The useCaseAvailable flag is only relevant for the client portion of a use case, because current use case availability of the server portion can already be discovered better with detailed discovery and reads / subscriptions to the corresponding features.
useCaseInformation. useCaseSupport. scenarioSupport (list)	UseCaseScenarioSupportType	0…unbounded	Within the scenarioSupport list all supported scenarios SHALL be stated with their corresponding scenario number.

2627 *Table 20: nodeManagementUseCaseData*

7.5.3 Partial Use Case Discovery

In most cases a device is only interested to discover matching use case actors. In this case a partial read can be performed on the nodeManagementUseCaseData function if partial read is supported by the nodeManagementUseCaseData function. In this case the “read. partial” elements is set within possible operations for the nodeManagementUseCaseData function in the nodeManagementDetailedDiscoveryData function.

Within the filter of the partial read the use case selectors allows to exactly specify parameters to match interoperability with other devices on the use case level.

The following elements MAY be used within nodeManagementUseCaseDataSelectors:

- useCaseInformation.actor
- useCaseInformation.useCaseSupport.useCaseName

Every device that supports partial use case discovery SHALL support the selectors “actor” and “useCaseName”.

7.5.4 Changes During Runtime

During runtime, a device MAY add/remove/modify one or more use cases. Therefore, a device that uses use case discovery should always subscribe to use case discovery.

8 Runtime behaviour

During runtime, which is the normal operation mode, a SPINE node is typically sending SPINE messages and receiving SPINE messages. The behaviour of a SPINE node is mostly specified within the corresponding SPINE resources (device type, entity type, feature type).

8.1 Runtime behaviour example (informative)

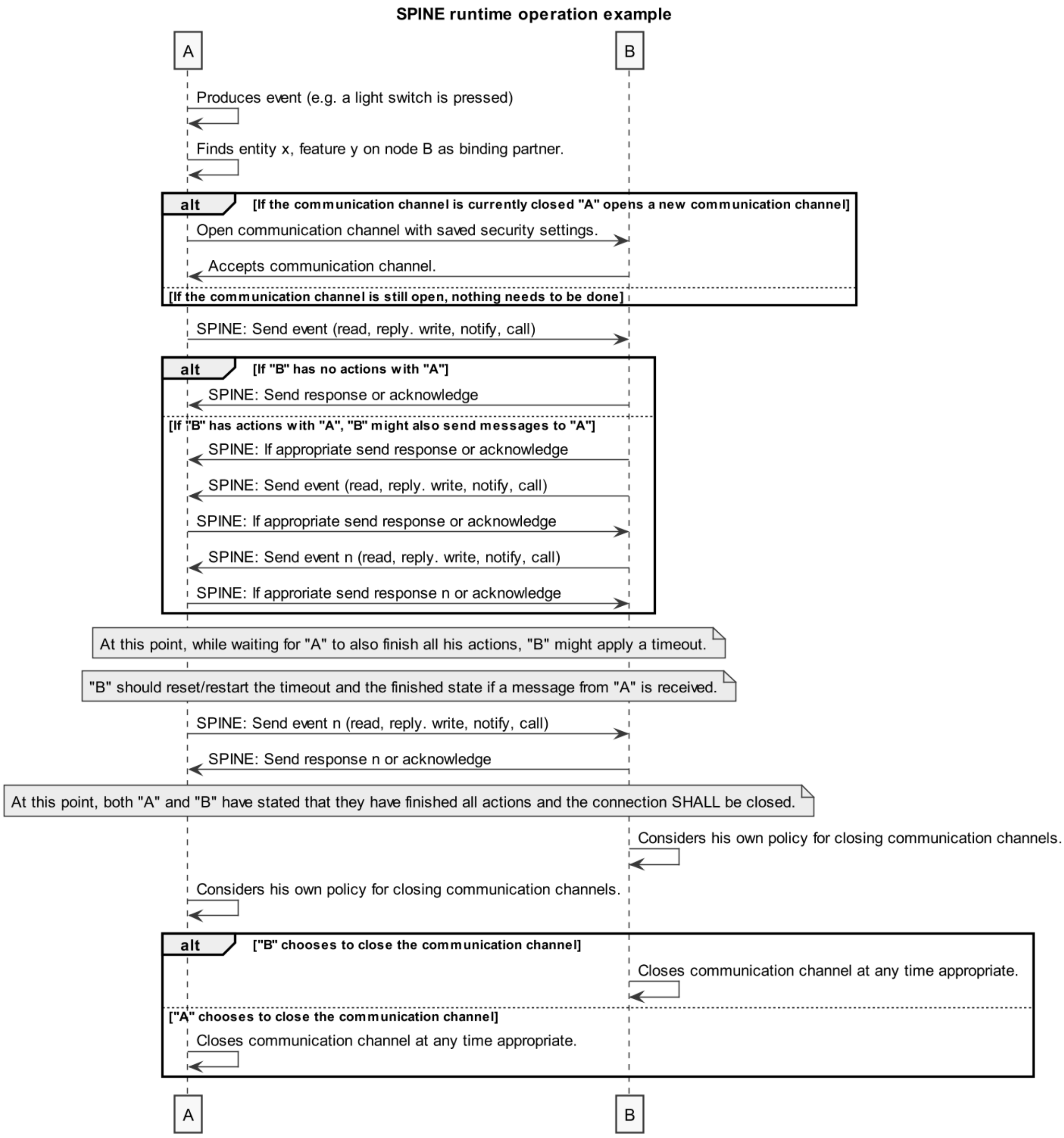
As an example, the following runtime is shown. First of all, we assume a detailed discovery between node A and B already took place.

Note: Some events shown in the picture take place in another layer and protocol than SPINE (e.g. "Open communication channel...").

In this example the light switch of node A is pressed. Furthermore, we assume node A has a specific binding for this case and produces a proper SPINE message (in this example a "toggle" write command). According to the binding it is sent to node B.

If node A already has an open communication channel to node B, e.g. because they haven’t yet closed their last open communication channel, then it can just go along and send events according to the feature type specification. However, if there is no open communication channel, it first has to open a communication channel.

Once all events based on the feature type specification have been sent, a connection termination handshake may be initiated.



2666

2667 *Figure 27: SPINE runtime behaviour example*

2668

Annex A - Recommendations for Restricted Function Exchange

The following considerations are not exhaustive! I.e. the restricted function exchange permits more cases than those considered in this section. However, it is recommended for standardised feature types based on standard classes to only use the combinations mentioned below. Complex classes and their corresponding feature types are out of scope of this section and may impose own recommendations.

- General rules that shall be considered:
  - Instead of only one element or one list entry, several operations on the same level(!) may be done in one message
  - <ELEMENTS> are only used for deletion or partial read
  - With ONLY cmdControl "delete" the <FUNCTION> must be empty
  - Within read-commands (classifier = read), <FUNCTION> must be empty
  - <SELECTORS> may only be stated if explicitly mentioned as rule
  - <ELEMENTS> may only be stated if explicitly mentioned as rule
  - <FUNCTION> must always be stated. In case of adding or modifying, the content must be stated in <FUNCTION>. In case of deletion, <FUNCTION> must be present, but empty!
  - Identifiers do not need to be stated in the <ELEMENTS> of a read function; identifiers must always be complete in a reply

In the following, applied rules and examples are denoted for the relevant combinations that may occur in implementations.

Classifier: **write**	
Adding content	**No List, element affected**
	Applied rules: - Element must not be present before.
	Examples: - EEBus_SPINE_Spec_Example_RFE_W-A-N-1-01.xml
	List, element affected in list entry
	Applied rules: - Element must not be present before - Identifier must be declared in <FUNCTION>
	Examples: - EEBus_SPINE_Spec_Example_RFE_W-A-Y_1-2-01.xml
	List, list entry affected
	Applied rules: - Identifier must not be present before - Identifier must be declared in <FUNCTION>
	Examples: - EEBus_SPINE_Spec_Example_RFE_W-A-Y_1-1-01.xml

Modifying content	**No List, element affected**
	Applied rules: - Element must be present before - Only the modified element must be stated in <FUNCTION>
	Examples: - EEBus_SPINE_Spec_Example_RFE_W-P-N-1-01.xml
	List, element affected in list entry
	Applied rules: - Element must be present before - Identifier and element must be declared in <FUNCTION>
	Examples: - EEBus_SPINE_Spec_Example_RFE_W-P-Y_1-1-01.xml
Deleting content	**No List, element affected**
	Applied rules: - Element must be present before - Element must be identified in <ELEMENTS>
	Examples: - EEBus_SPINE_Spec_Example_RFE_W-D-N-1-01.xml
	List, element affected in list entry
	Applied rules: - Element must be present before - List entry must be present before - Identifier must be declared in <SELECTORS> - Element must be identified in <ELEMENTS>
	Examples: - EEBus_SPINE_Spec_Example_RFE_W-D-Y_1-2-01.xml
	List, list entry affected
	Applied rules: - List entry must be present before - Identifier must be declared in <SELECTORS>
	Examples: - EEBus_SPINE_Spec_Example_RFE_W-D-Y_1-1-01.xml
	No List, element affected

Adding, **modifying** and **deleting** content	*Applied rules:* - For non-list functions add AND modify AND delete of element(s) is possible within one message - For Add: Element must not be present before; the new element is stated in <FUNCTION> - For Modify: Element must be present before; the modified element is stated in <FUNCTION> - For Delete: Element must be present before; the element is stated in <ELEMENTS>
	Examples: - EEBus_SPINE_Spec_Example_RFE_W-M-N-1-01.xml
	List, element affected in list entry
	Applied rules: - Elements in one list entry may be added AND modified AND deleted within one message - For Add: Element must not be present before; new element and the identifier must be stated in <FUNCTION> - For Modify: Element must be present before; modified element and the identifier must be stated in <FUNCTION> - For Delete: Element must be present before; element to delete must be stated in <ELEMENTS> and must not be stated in <FUNCTION>; the identifier must be stated in <SELECTORS>
	Examples: - EEBus_SPINE_Spec_Example_RFE_W-M-Y_1-2-01.xml
	List, list entry affected
	Applied rules: - A complete list entry may be added OR modified OR deleted - For Add: Identifier must not be present before; complete entry must be stated in <FUNCTION> - For Modify: Identifier must be present before; complete entry must be stated in <FUNCTION>; no new elements may be added or existing ones deleted, only existing ones may be modified - For Delete: Identifier must be present before; identifier is selected via <SELECTORS>
	Examples: - EEBus_SPINE_Spec_Example_RFE_W-M-Y_1-1-01.xml (add) - EEBus_SPINE_Spec_Example_RFE_W-M-Y_1-1-02.xml (modify) - EEBus_SPINE_Spec_Example_RFE_W-M-Y_1-1-03.xml (delete)
Classifier: **notify**	
Added content	**No List, element affected**

	Applied rules: - All required information is given in <FUNCTION> only - Only the newly added element shall be stated
	Examples: - EEBus_SPINE_Spec_Example_RFE_N-A-N-1-01.xml
	List, element affected in list entry
	Applied rules: - All required information is given in <FUNCTION> only - Only the newly added element together with the identifier of the list entry shall be stated
	Examples: - EEBus_SPINE_Spec_Example_RFE_N-A-Y_1-2-01.xml
	List, list entry affected
	Applied rules: - All required information is given in <FUNCTION> only - Only the newly added list entry shall be stated
	Examples: - EEBus_SPINE_Spec_Example_RFE_N-A-Y_1-1-01.xml
Modified content	*Applied rules:* - All the same as for [**Added** content], but with modified elements instead of new ones
	Examples: - See [**Added** content]
Deleted content	**No List, element affected**
	Applied rules: - No <SELECTORS> or <FUNCTION> - The deleted element is defined within the <ELEMENTS>
	Examples: - EEBus_SPINE_Spec_Example_RFE_N-D-N-1-01.xml
	List, element affected in list entry
	Applied rules: - The identifier of the list entry, where the element was deleted is stated in <SELECTORS> - The deleted element is defined within the <ELEMENTS>
	Examples: - EEBus_SPINE_Spec_Example_RFE_N-D-Y_1-2-01.xml
	List, list entry affected

	Applied rules: - The identifier of the deleted list entry is stated in <SELECTORS>
	Examples: - EEBus_SPINE_Spec_Example_RFE_N-D-Y_1-1-01.xml
Added, **modified** and **deleted** content	**No List, element affected**
	Applied rules: - For non-list functions a notification of added AND modified AND deleted element(s) is possible within one message - For Add: New element is stated in <FUNCTION> (not in <ELEMENTS>) - For Modify: Modified element is stated in <FUNCTION> - For Delete: Deleted element is stated in <ELEMENTS> and must not be stated in <FUNCTION>
	Examples: - EEBus_SPINE_Spec_Example_RFE_N-M-N-1-01.xml
	List, element affected in list entry
	Applied rules: - Elements in one list entry may be added AND modified AND deleted within one message - For Add: New element together with the identifier is stated in <FUNCTION> - For Modify: Modified element together with the identifier is stated in <FUNCTION> - For Delete: Deleted element is stated in <ELEMENTS> and must not be stated in <FUNCTION>; identifier is stated in <SELECTORS>
	Examples: - EEBus_SPINE_Spec_Example_RFE_N-M-Y_1-2-01.xml
	List, list entry affected
	Applied rules: - A complete list entry may be added OR modified OR deleted - For Add: New list entry is stated in <FUNCTION> - For Modify: Modified list entry is stated in <FUNCTION> - For Delete: Identifier is stated in <SELECTORS> but not in <FUNCTION>
	Examples: - EEBus_SPINE_Spec_Example_RFE_N-M-Y_1-1-01.xml (added) - EEBus_SPINE_Spec_Example_RFE_N-M-Y_1-1-02.xml (modified) - EEBus_SPINE_Spec_Example_RFE_N-M-Y_1-1-03.xml (deleted)
Classifier: **read**	
Reading **partial** content	**No List, element affected**
	Applied rules: - The element that shall be read is stated in <ELEMENTS>

	Examples: - EEBus_SPINE_Spec_Example_RFE_RD-P-N-1-01.xml
	List, element affected in list entry
	Applied rules: - The identifier that defines the list entry that shall be read is stated in <SELECTORS> - The element that shall be read is stated in <ELEMENTS>
	Examples: - EEBus_SPINE_Spec_Example_RFE_RD-P-Y_1-2-01.xml
	List, list entry affected
	Applied rules: - The identifier that defines the list entry that shall be read is stated in <SELECTORS>
	Examples: - EEBus_SPINE_Spec_Example_RFE_RD-P-Y_1-1-01.xml
Classifier: **reply**	
Replying **partial** content	**No List, element affected**
	Applied rules: - All required information is given in <FUNCTION> only
	Examples: - EEBus_SPINE_Spec_Example_RFE_RY-P-N-1-01.xml
	List, element affected in list entry
	Applied rules: - All required information is given in <FUNCTION> only
	Examples: - EEBus_SPINE_Spec_Example_RFE_RY-P-Y_1-1-01.xml

Table 21: Applied RFE rules and example overview

Annex B - Access limitations

It is common practice to impose restrictions on the access of a device's data and functionality. This is most important to keep private information secret and also to ensure safety aspects. However, as this specification does not require a specific communications technology, there is no dedicated authentication or other security/safety concept or something comparable imposed so far. Instead, the technology specific security mechanisms can be mapped to a generic trust level information. From this point of view a trust level makes security mechanisms of different technologies comparable over generic information. Please also note some restrictions may originate from legal issues rather than communications technologies.

This specification uses the term "trusted nodes" for two device's information exchange, without specifying in detail how this "trust" is gained. However, some aspects shall briefly be explained. Let's consider the commissioning of two devices:

Step 1: Trust node with general or specific release of view

Some kind of commissioning is required in order to establish a connection with another node and to keep this node as trusted communication partner. Trusting the other node includes granting access to the own primary NodeManagement instance. However, as subsequently described the content of the NodeManagement instance may differ dependent on the communication partner and commissioning process.

In general, a node SHOULD offer its NodeManagement instance with all entities and features that are required for normal operation and esp. for interoperable processes. This situation is the basis for the underlying protocol specification and is subsequently called a "normal view".

Only for specific security requirements (e.g. if a certain trust level is not given for access to the corresponding feature) or non-interoperable processes the commissioning step MAY lead to a reduced set of entities and features in the NodeManagement instance.

Step 2: Consider trust level

In either case of "step 1" the NodeManagement instance contains all entities and features whose presence can be published to the communication partner. However, this does not necessarily mean that access to the features (i.e. data exchange) is possible as well. The NodeManagement instance can include elements called "minimumTrustLevel" that report which kind of "trust level" is required at least in order to not just know the presence of the feature, but also to exchange data with it.

Dependent on the communications technology the commissioning phase of "step 1" should already define a trust level for the kind of the commissioning. This may result in the trust with another node that has then not a sufficient trust level for all features. However, the communications technology may provide mechanisms to adjust (esp. increase) the trust level later on.

Summary

A “special kind” of commissioning or special commissioning parameters of step 1 may lead to a different kind of view on the own primary NodeManagement instance (i.e. the set of entities and features presented to the other node). Changing the view would typically require the deletion of the view with the other node (which may require to “forget” the other node and perform a commissioning from scratch again) and the execution of a new (and different) “detailed discovery” afterwards.

Please note that the view should only be considered as some kind of “filter”. Even with a given view, the set of entities and features may vary because firmware upgrades enhance a device’s functionalities, e.g.

In contrast to the view, the access to features may vary more dynamically. I.e. a node may enhance its trust level towards its communication partner later on.

Annex C - Release versioning

C.1 Introduction

This section defines rules regarding the versioning of SPINE releases.

All SPINE specification documents underlie a joint versioning. SPINE Protocol Specification, SPINE Resource Specification and SPINE XSDs are always versioned with the same version number. For SPINE Protocol Specification and SPINE Resource Specification, the version number is stated on the front page of the Specification documents. For official SPINE XSDs the corresponding XML namespace contains only the major version number (see section 4.3.4.3) and the "version" attribute contains the complete version number (see section 4.3.4.4).

For SPINE releases a version number with 3 numerals, separated by the character ".", shall be used.

C.2 Rules

C.2.1 Compatibility aspects

The subsequent sections use the term "downward compatibility" and describe in which situations this compatibility can be expected. For more details on compatibility aspects please consider chapter 4.

C.2.2 Final releases

A final ("official") release is based on the results from the specification phase, like described in section C.2.3. It is not allowed that there are any technical changes between the last release candidate from the specification phase and the final release. Only minor editorial corrections are permitted.

The specification version number is constituted in the following order:

V(major number).(minor number).(revision number)

Examples:

- SOME_specification_V1.0.0.pdf
- SOME_specification_V1.5.13.pdf

C.2.2.1 Major

Major releases are addressed over the 1st numeral in the version number. Only a new major release is allowed to break downward compatibility. Also, the EEBus Initiative e.V. will try its best to preserve downwards compatibility as long as possible. With technical progression there might be a time where another major release needs to break downward compatibility.

C.2.2.2 Minor

Minor releases are addressed over the 2nd numeral in the version number. A certain amount of revision releases or a bigger extension may lead into a minor release. All changes in a minor release must be downward compatible.

C.2.2.3 Revision

Revision releases are addressed over the 3rd numeral in the version number. An example for a revision release is a new requirement that arises on the side of one or more EEBus members that needs to be addressed while maintaining downwards compatibility.

C.2.3 Specification phase

Depending on the requirements, use cases and ideas, a specification phase is started. For the specification phase the interested members form a working group and define an interoperable future proof solution.

In contrast to final releases, the specification phase describes the time and process between official releases. This typically includes intermediate releases which are described subsequently.

C.2.3.1 Draft/alpha/beta

During specification phase the results of the working group are considered as draft version. As the specification phase is in most cases also a finding phase, there may still occur drastic changes from version to version. Therefore, versions in the draft state need not be downward compatible. However, it is in the own interest of all participants especially during the final phase of specification, that big changes, compared to previous versions, are also underlined with strong reasons.

All versions that append a “draft”, “alpha” or “beta” behind the version number are considered as draft/alpha/beta versions. Draft/alpha/beta versions must always append a numeral that is incremented with each successive draft, alpha or beta version.

Note: For documents it is more common to use “draft”, for a software it is more common to use “alpha” or “beta”, depending on the progress. Therefore, “draft” shall be used for documents and “alpha” or “beta” for everything else.

Examples:

- SOME_specification_V1.0.0_draft15.pdf
- SOME_specification_V1.0.0_alpha15.zip
- SOME_specification_V1.0.0_beta15.zip

C.2.3.2 Release candidate

One of the last steps of a specification phase is to commit to a first release candidate. The release candidate shall be binding for a proof-of-concept phase. Findings during a proof-of-concept phase or during a commenting phase might still lead to additional release candidates, if necessary.

All versions that append a “RC” behind the version number are considered release candidates. Each release candidate shall also append a numeral (e.g. RC1 or RC2), that is incremented with each successive release candidate.

Example:

- SOME_specification_V1.0.0_RC2

C.2.3.3 Snapshot releases

It is sometimes helpful to create a snapshot of the current specification development. Such snapshots typically occur between draft/alpha/beta releases, but they may also occur between two release candidates. A snapshot is “less formal” than any of the other releases. However, even snapshot releases create some effort for creation and maintenance and should only be considered if necessary.

The name of a snapshot release always consists of the next scheduled non-snapshot release, followed by the string "snapshot" and the current repository revision (each separated with an underscore).

Example:

- Last release name: SOME_specification_V1.0.0_beta5
  In this example we assume that “SOME_specification_V1.0.0_beta5” was created from the repository revision “1498”.
- Next (not yet finished) release name: SOME_specification_V1.0.0_beta6
  In this example we assume that the development towards “SOME_specification_V1.0.0_beta6” began with the repository revision “1501”.
- Current repository (e.g. SVN) revision: 1539
- Name of snapshot release: SOME_specification_V1.0.0_beta6_snapshot_1539

C.2.3.4 Release date

The release date (written in the format "YYYYMMDD") MAY be included in the release name between the version information and a label (see section C.2.3.5), separated by an underscore:

- In case of a final release: after the version. Example:
  - SOME_specification_V1.0.0_20150522
- In case of a draft/alpha/beta/release candidate release: after the number directly behind the (numbered) string draft, alpha, beta or RC. Examples:
  - SOME_specification_V1.0.0_draft5_20150522
  - SOME_specification_V1.0.0_alpha8_20150522

    - SOME_specification_V1.0.0_beta3_20150522
    - SOME_specification_V1.0.0_RC1_20150522
- In case of a snapshot release: after the repository revision info. Example:
    - SOME_specification_V1.0.0_beta7_snapshot_1784_20150522

C.2.3.5 Labelling

A release may have a label, added as the last part of the release name (separated with an underscore). It SHALL NOT include one of the following:

- The string "snapshot"
- Any number (unless the label is a name and number intrinsically belongs to the name, like “3” intrinsically belongs to “W3C”, e.g.)

The label shall ONLY consist of alphanumerical characters and the following characters:

    - -
    - (
    -)

No other character is permitted.

Examples:

- SOME_specification_V1.0.0_draft5_lastDraftForThisYear
- SOME_specification_V1.0.0_alpha8_20150522_temp-release(for-vendor-XYZ)
- SOME_specification_V1.0.0_beta3_(fairABC-release)
- SOME_specification_V1.0.0_RC1_20150522_first-release-candidate
- SOME_specification_V1.0.0_beta7_snapshot_1784_20150522_ProjectX-InterRelease
- SOME_specification_V1.0.0_final

C.2.4 Overview

The previously defined rules lead to a versioning, like shown in the following figure.

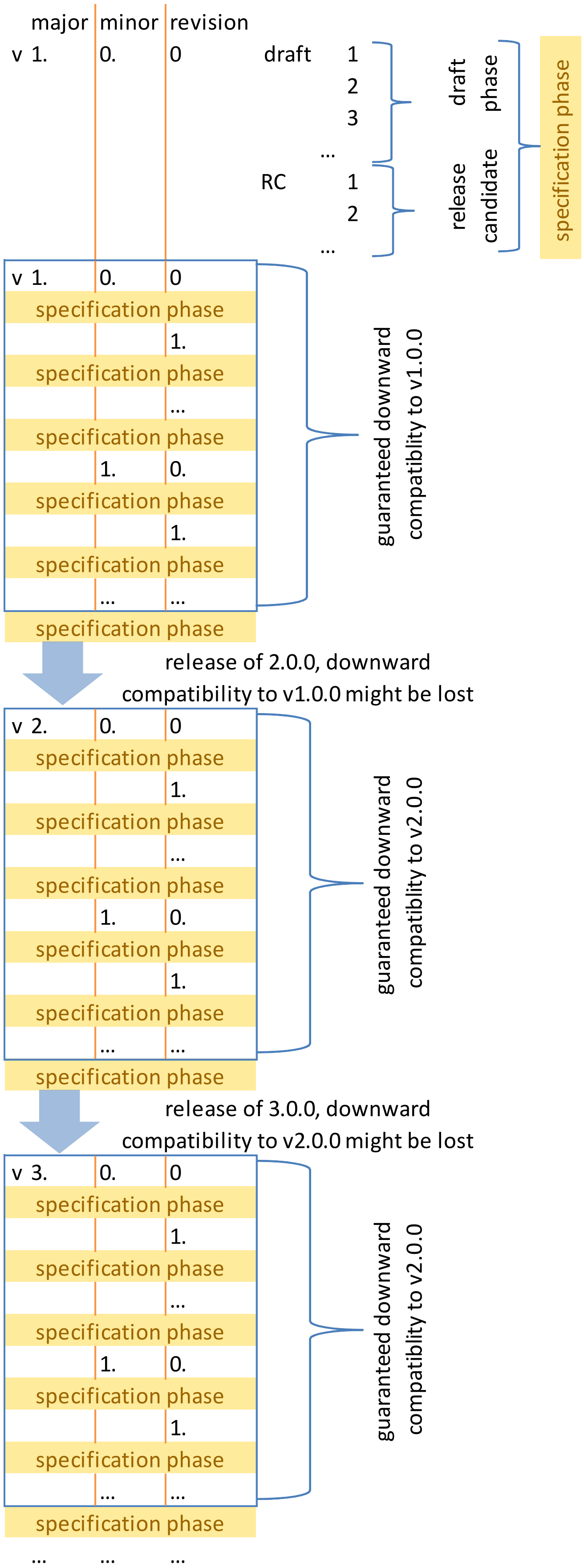


Figure 28: Graphical explanation of SPINE release versioning

Annex D – Recommendations for initial connections

D.1 Introduction

The SPINE layer can be used with different communications technologies. Some communications technologies may have some kind of "automatic connection process" between two devices. In such cases the question arises whether it is possible to conclude on application level whether such an automatic connection is "useful" (i.e. the connection should be continued) or whether the connection is "not useful" (i.e. the connection can or even should be closed in order to focus on connections with another device). The next sections give some recommendations for the SPINE layer to provide for a reasonable communication behaviour.

D.2 Terms

A "functional communication" means that a client feature of one device uses a functional server feature of the other device. Here, "functional server feature" means a feature different than the primary NodeManagement instance.

A "well-known functional communication partner" is a communication partner that is already known as device to have "functional communication" with.

An "undetermined functional communication partner" is a communication partner that is not recognized as "well-known functional communication partner" so far.

D.3 Recommendations

This section considers a device "A" and a device "B" that just set up a connection.

If the devices already recognize at this point the respective communication partner as "well-known functional communication partner", then the communication should be continued to permit "functional communication". Of course, any device may terminate the communication if required. However, the "well-known functional communication" is considered as "normal communication" and is NOT considered further in this section.

If a device does NOT know the SPINE address of the communication partner at this point, it should send a detailed discovery request to the communication partner in order to get the SPINE address. When it receives the SPINE address it should re-evaluate whether the communication partner is a "well-known functional communication partner" as described above.

Subsequently it is assumed the communication partner is considered as "undetermined functional communication partner":

The following recommendations are given with the focus on features with role "server". We consider a device "A" that has one or more server features:

1. If device “A” DOES NEITHER receive a “proper command” for at least one of its server features NOR a “detailed discovery request” from the communication partner within 30 seconds, then it may consider the connection as “not useful” and close the connection.
2. If it just receives a “detailed discovery request” within the above-mentioned time frame: Device “A” should perform item 1 again with a new (i.e. reset) time frame.
3. If it receives a “proper command” within the above-mentioned time frame: Device “A” should finally consider the communication partner as “well-known functional communication partner”.

The term “proper command” applies in ANY of the following cases:

1. The command was sent to a server feature of device “A” and is a valid for this server feature.
2. The command is a valid binding request or subscription request for one of device “A”’s server features.

The following recommendations are given with the focus on features with role “client”. We consider a device “A” that has one or more server features:

1. A device “B” with client features should consider the probable behaviour of a device “A”: This means device “A” may consider the connection as “not useful” if device “A” does not receive a “proper command” in time.
2. If device “B” does not find a matching server feature at device “A” it may consider the connection as “not useful”.

Annex E – Examples of enhanced communication mode and DestinationList

E.1 Introduction

The subsequent sections are informative only! They show how the enhanced communication mode across multiple devices is meant to work and how this is related with DestinationList. This is explained with some example architectures of devices. Please note that these examples are just informative and are not meant to impose specific or additional requirements for an implementation.

E.2 Technology gateway types

There are basically two typical types of technology gateways:

1. Gateway type 1: Vendor specific
   Exports (a part of) a vendor specific system (devices or system functionalities) to a common communication/application protocol (e.g. SPINE over SHIP, subsequently called "SHIP-SPINE").
2. Gateway type 2: Common technology bridge
   Integrates several common communication/application protocols (at least partly, for example to enable a specific functionality).

These gateway type descriptions should not be interpreted as definitions. They are just used to explain typical approaches in this chapter.

Figure 29 shows an example for a system with a vendor specific system that exports some functionality into a SHIP-SPINE system.

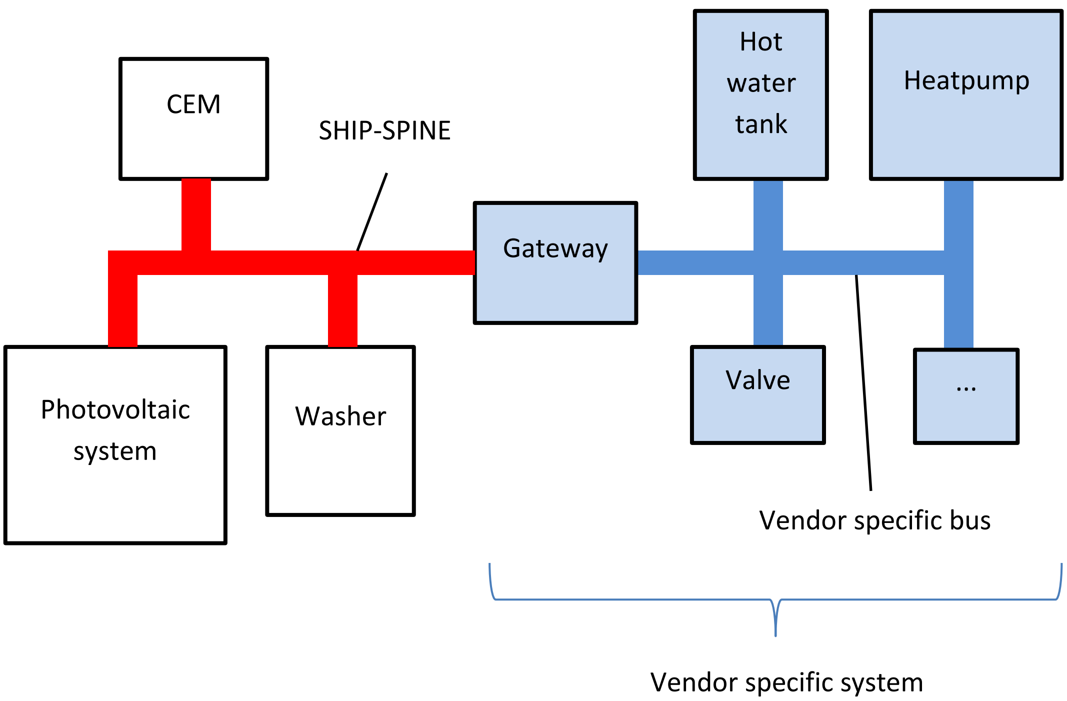


Figure 29: Gateway type 1 example for vendor specific systems together with a SHIP-SPINE system

The SHIP-SPINE system uses SHIP for the physical communication management and the transport of SPINE messages.

There are different options for the gateway type 1 with regards to the SHIP-SPINE system:

1. The vendor's gateway does not offer any device of the vendor specific system as SPINE device directly. Instead, the gateway aggregates the vendor specific system (at least a chosen set of functionalities) into a single "virtual" device and offers the result as a single SPINE device.
2. The vendor's gateway offers ("exports") some devices of the vendor specific system as distinct SPINE devices (each with a chosen set of functionalities).
3. The vendor's gateway does not offer any device of the vendor specific system as SPINE device directly. Instead, the gateway aggregates the vendor specific system (at least a chosen set of functionalities) into several "virtual" devices and offers them as distinct SPINE devices.

In Figure 30 an example is sketched with a gateway type 2 that integrates at least some functionalities of several common communications technologies into a combined SPINE system with some SHIP-devices and a CoAP-based device.

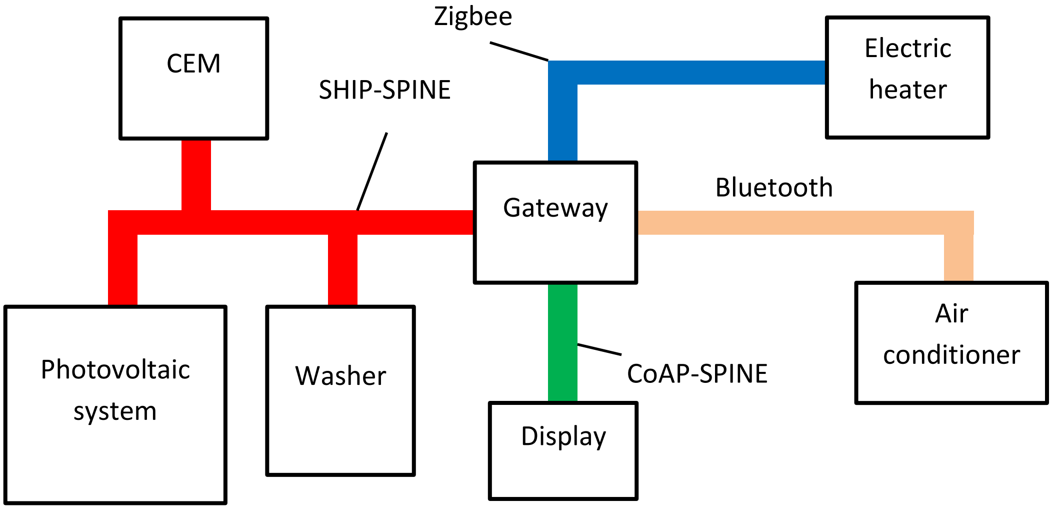


Figure 30: Gateway type 2 example for common protocols together with two SPINE (sub-)systems.

For gateway type 2 the same options apply as described above for gateway type 1.

Each communications system supported by a gateway is subsequently referred to as "interface". This means the gateway of Figure 29 has two interfaces whereas the gateway of Figure 30 has four interfaces. In both gateway type examples similar procedures need to be applied in the gateway if a "native" SPINE device "A" (a SHIP-SPINE device, e.g.) submits a message to a device "B" of a different interface or conversely receives a message from it. Of course, this is only feasible if the gateway offers device "B" as "SPINE" device. The next section discusses some of these aspects.

E.3 Provision of DestinationList for message forwarding

As mentioned before there are similar options for the gateway types and similar procedures required if a message involves communication across distinct interfaces. This will be discussed now with a more generic example shown in Figure 31.

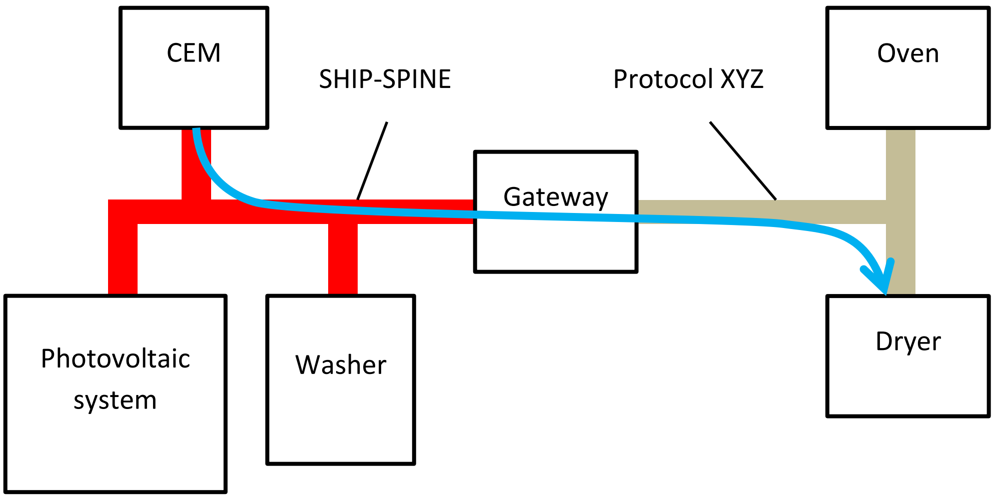


Figure 31: More generic gateway example with the interfaces "SHIP-SPINE" and "Protocol XYZ".

In this example the device "CEM" will submit a SHIP-SPINE message to the device "Dryer". Before this can take place, some preconditions and steps need to be fulfilled. First of all, the gateway needs to satisfy these preconditions:

1. The gateway can communicate with "Dryer":
    a. The gateway has an own "XYZ address" (i.e. an address it has within the "Protocol XYZ" system).
    b. The gateway knows the "XYZ address" of "Dryer" (i.e. the address that "Dryer" has within the "Protocol XYZ" system).
2. The gateway can represent "Dryer" as SPINE device (at least partially):
    a. The gateway can map at least some of the "Dryer" functionalities into a proper SPINE functionality.
    b. The gateway assigns for "Dryer" a SPINE address.
    c. Based upon the previous items the gateway creates internally a SPINE representation of "Dryer", comprising of a proper primary NodeManagement instance and related Entities and Features for "Dryer".
3. The gateway can represent itself as SPINE device:
    a. The gateway has an own SPINE device address.
    b. The gateway implements a primary NodeManagement instance and related Entities and Features about itself.
4. The gateway can offer "Dryer" as SPINE device:
    a. Based upon item 2 the gateway adds SPINE information about "Dryer" into the "DestinationList" function of the gateway's own primary NodeManagement instance.
5. The gateway can communicate with "CEM":
    a. The gateway has an own SHIP address (for the physical communication).

b. From a detailed discovery request initiated by "CEM" the gateway knows the SHIP address of "CEM" (due to the received SHIP message) and its SPINE (from the SPINE content of the SHIP message) address. Conversely, "CEM" knows from the proper reply of the gateway the SPINE address of the gateway.

Afterwards, the "CEM" needs to get knowledge of "Dryer":

1. The "CEM" evaluates the detailed discovery reply of the gateway. Among others, the detailed discovery of this example contains information that the gateway has a DestinationList.
2. The "CEM" requests and evaluates the DestinationList of the gateway. In this example it contains an entry with the SPINE address of "Dryer".

From this moment onwards "CEM" knows that messages for "Dryer" need to go through the gateway. This is already the case if "CEM" wants to get a detailed discovery of "Dryer" (as "CEM" just knows the SPINE address of "Dryer" and nothing else about "Dryer" yet):

1. "CEM" creates a SPINE message to read the detailed discovery of "Dryer". In the header of this SPINE message the major address elements are set as follows: "CEM" sets its own SPINE address into the "addressSource" element and sets the SPINE address of "Dryer" into the "addressDestination" element.
2. As "CEM" knows that "Dryer" can only be accessed via the gateway, "CEM" submits this SPINE message via SHIP to the SHIP address of the gateway.
3. In the received SHIP message the gateway extracts the SPINE message and encounters from the field "addressDestination" that this SPINE message is intended for "Dryer".

The next section discusses details on the "forwarding" of the received message.

E.4 Interfaces and "internal routing"

This section briefly discusses the relation between physical interfaces (and the related protocols), SPINE addresses and the DestinationList instance of the gateway.

E.4.1 General concept

Figure 32 shows an example architecture for the gateway of Figure 31.

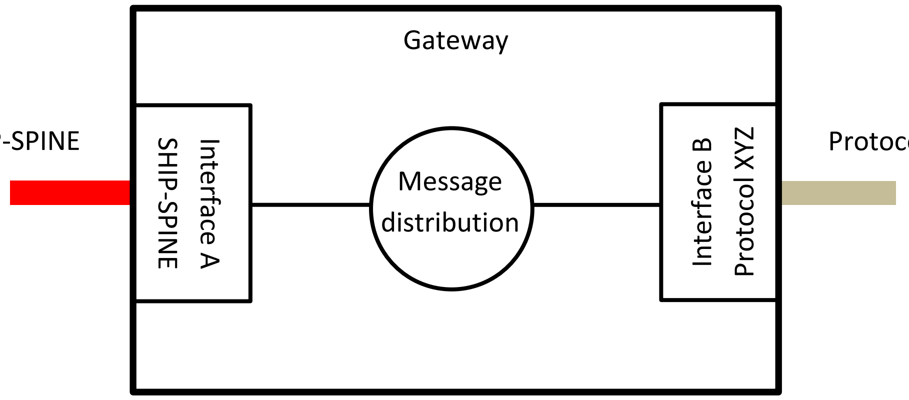


Figure 32: Example schema for gateway interfaces and message distribution

Internally, the gateway may keep a relation as shown in Table 22.

Device	Interface	Protocol address of device	SPINE address origin of device
CEM	A	SHIP address	Received from device
Photovoltaic system	A	SHIP address	Received from device
Washer	A	SHIP address	Received from device
Oven	B	Protocol XYZ address	Created by gateway for device
Dryer	B	Protocol XYZ address	Created by gateway for device

Table 22: Relation between connected devices, interfaces and addresses

This way the gateway can associate physical addresses with proper SPINE addresses. In this example, the devices attached at interface B are those the gateway puts into its DestinationList instance.

Section E.3 gave an example that finished with a message received from "CEM" with "Dryer" as destination. This is discussed a bit in more detail now.

Technically, the gateway receives a SHIP message. From the SHIP layer it can only know that the message stems from "CEM" and that it was submitted to the gateway. From the payload of the SHIP message the gateway can extract the SPINE message of "CEM". In this example, the field "addressDestination" is set to a value that does not contain the SPINE address of the gateway itself. instead, the gateway encounters a match with the SPINE address of "Dryer" from its DestinationList instance. From Table 22 it knows that this message now needs to be processed by interface B with proper "Protocol XYZ" conformant interactions with "Dryer".

The completion of these interactions with "Dryer" will be a simple positive or negative acknowledgement (if the SPINE message of "CEM" contained a "write" command, e.g.) or a data response from "Dryer" (if the SPINE message of "CEM" contained a "read" command, e.g.). In either case the gateway can take the result from interface B and create a proper SPINE message as response to "CEM". In this response, the field "addressSource" would be set to the SPINE address of "Dryer". Then, this SPINE message can be submitted in a SHIP payload to "CEM".

E.4.2 Combination of SPINE networks

In Figure 30 a gateway example was given where two SPINE networks were attached: One using "SHIP-SPINE" and another one using "CoAP-SPINE" (i.e. SPINE over CoAP). Although section E.4.1

remains valid for the connection of different interfaces the situation simplifies between such SPINE networks.

Table 23 shows the relation for this example, with interfaces A to D for SHIP-SPINE, Zigbee, Bluetooth and CoAP-SPINE (in this order).

Device	Interface	Protocol address of device	SPINE address origin of device
CEM	A	SHIP address	Received from device
Photovoltaic system	A	SHIP address	Received from device
Washer	A	SHIP address	Received from device
Electric heater	B	Zigbee address	Created by gateway for device
Air conditioner	C	Bluetooth address	Created by gateway for device
Display	D	CoAP address	Received from device

Table 23: Relation between connected devices, interfaces and addresses for Figure 30

Forwarding a message between SPINE interfaces (in this case between interfaces A and D) is usually simpler in general as in such a case the gateway does not need to create an internal full representation of the devices of the respective other interface (though there might be a need to keep some communication states, e.g.).

However, in this case a gateway needs to create and offer two different DestinationList instances. Towards devices of interface A it would contain entries for "Electric heater", "Air conditioner" and "Display". Towards devices of interface D it would contain entries for "CEM", "Photovoltaic system", "Washer", "Electric heater" and "Air conditioner".

E.5 Access "simple" devices via SPINE proxy

A device with a networkFeatureSet value "simple" is intended for devices with limited communication capabilities that permit only the "simple communication mode", as explained in chapter 6 and esp. section 6.1. With regards to SPINE concepts, this usually means a "simple" device is "directly connected" to another SPINE device, i.e. without any other SPINE device in between.

Examples as shown in sections E.2 and E.3 make use of the enhanced communication mode. This means "simple" devices cannot be integrated in the same way because they just support the simple communication mode. However, with some additional implementation effort a technology gateway can well put "simple" devices into its DestinationList (see section 7.2.3.1) and can "somehow forward" messages to a "simple" device. The well-known concept of a "web proxy" (or "proxy server") can be used for this purpose to derive a similar "SPINE proxy" concept in this section. Please note this concept is intended to enable access from an enhanced communication mode capable device to a "simple" device; i.e. this concept is NOT intended to enable that a "simple" device can "find" or "see" other devices than its directly connected communication partner.

Figure 33 shows an example setup where the technology gateway acts as SPINE proxy between a SHIP-SPINE display and a "Protocol XYZ" based sensor. Before some technical details are discussed, please note a "gateway" is not obliged in general to put "simple" devices into its DestinationList. Consequently, the presence or absence of any SPINE proxy implementation remains a vendor specific decision (unless specific rules impose an implementation). Anyhow, the subsequent explanations shall help to understand the essential details for such an implementation.

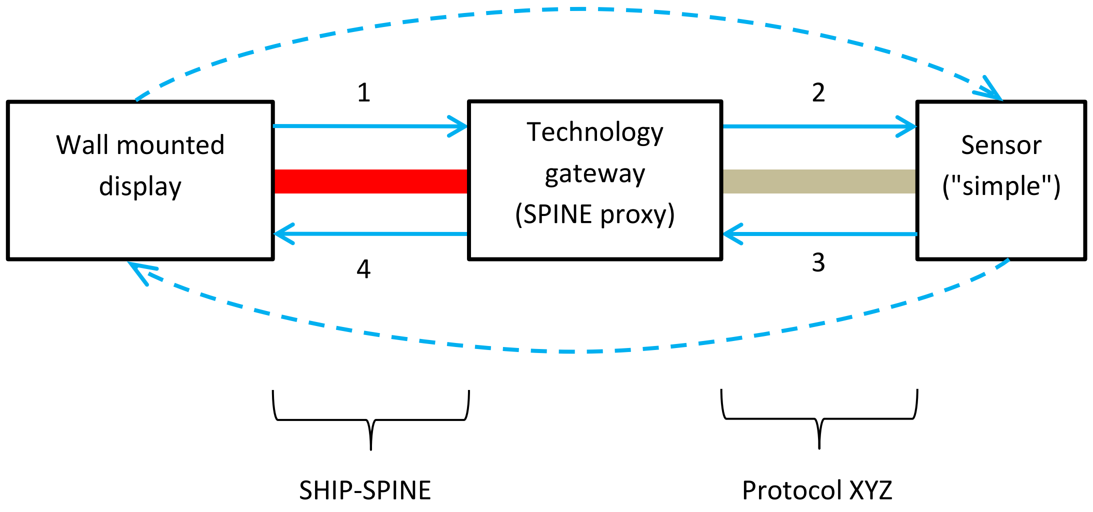


Figure 33: Example setup with a technology gateway acting as SPINE proxy to enable "access" to a "simple" device. Four message instances are exchanged in this example (solid arrows). The dashed arrows show the display's intention in terms of a SPINE message exchange.

Basically, a SPINE proxy needs to create an own message based upon a received message. This is required at least to adjust address information (just to give a comparison: a typical web proxy "hides" the requester's address this way). The SPINE proxy also needs to buffer the original request in order to create later on a proper reply once the "simple" device responds. This principle is shown with the following example.

The gateway receives a SHIP message (1) from the display and extracts the SPINE payload.

Message 1	
Received SPINE data	
Element	**Value**
addressSource	SPINE address of the display
addressDestination	SPINE address of the sensor
msgCounter	1114
msgCounterReference	- (not set)
payload	SPINE command of display

Table 24: Example for message 1 of Figure 33.

As the sensor is mentioned as destination the gateway begins with a proper interaction at the "Protocol XYZ" interface. In this example we assume the requested SPINE functionality requires just a single "Protocol XYZ" message towards the sensor (message 2) and only a single "Protocol XYZ" response from the sensor (message 3). Regardless of the details of "Protocol XYZ", the gateway needs to create message 2 as a new message that contains no address relation to the display. This means even if "Protocol XYZ" is another SPINE capable channel (CoAP-SPINE, e.g.) the related SPINE message 2 towards the sensor

1. would contain the gateway's SPINE address in the addressSource field (not the SPINE address of the display) and
2. would contain in "msgCounter" a new value (3001, e.g., but usually not 1114).

Once message 3 is received at the gateway, the gateway needs to create message 4

1. with payload based upon message 3, but
2. with header fields set in relation to message 1.

This is also the case if "Protocol XYZ" is another SPINE capable channel.

Message 4	
SPINE data, response to message 1	
Element	**Value**
addressSource	SPINE address of the sensor
addressDestination	SPINE address of the display
msgCounter	377 (example value; to be chosen by the gateway)
msgCounterReference	1114
payload	SPINE command derived from message 3

Table 25: Example for message 4 of Figure 33.

The implementation effort for a SPINE proxy to export a "simple" device increases if binding (see section 7.3) or subscription (see section 7.4) need to be considered: Regardless whether Protocol XYZ is a SPINE capable protocol or not the "simple" device communicates with the SPINE proxy only, i.e. the "simple" device may only support subscription or binding functionality towards the SPINE proxy. This means that each subscription or binding request from any of the SHIP-SPINE devices must be managed by the SPINE proxy completely and will not be forwarded to the "simple" device as this was shown for other messages in Figure 33. I.e. it is up to the SPINE proxy implementation only to decide whether it permits binding at all for a requested feature of the "simple" device. And if it provides this functionality

1. it can decide whether it permits just one or more bindings for this feature and
2. it has to reject "write" operations to this feature if this feature requires a binding but the originator of the "write" request has no valid binding.

In fact, every SPINE device with "binding" functionality for its features has the same decision options. But if a SPINE proxy provides access to a "simple" device it is the SPINE proxy that would need to implement the management of the "binding" functionality. With regard to "subscription" functionality, this is similarly the case.

E.6 Forwarding to "next hop"

Especially sections E.2 and E.3 focus on technology gateway concepts. Although sophisticated router capabilities have been postponed to future versions of SPINE, a possible functionality will be shown for devices with the element networkFeatureSet set to "router". Such devices typically just connect other devices of the same communications technology. However, even in this case there may be more than just one interface where devices are attached to. In general, this means the example architecture in Figure 32 can be used for "router" devices as well.

The examples shown so far focus on only one intermediate device, i.e. a gateway. Figure 34 shows an example where three intermediate devices are required for the communication between the display and the washer.

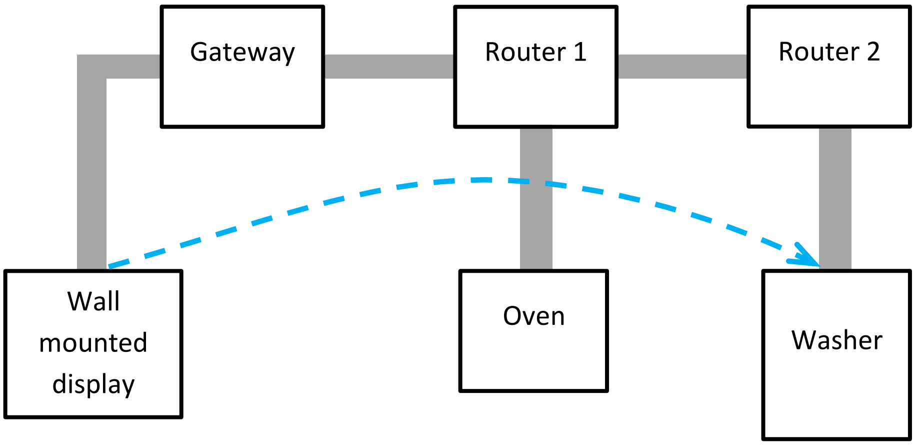


Figure 34: Example setup to demonstrate the "next hop" principle: "Gateway" needs to know that "Router 1" is the next hop to reach "Washer".

Tables like Table 22 and Table 23 are useful for devices that are directly attached to the respective interface. However, for situations like in Figure 34 it is more useful to consider an additional table with a "next hop" information: "Gateway" can keep in its table that the "Router 1" device is the "next hop" to reach "Washer". Similarly, "Router 1" can keep in its table that "Router 2" is the "next hop" for "Washer".

E.7 Network aspects

The example scenarios of the previous sections used DestinationList instances to export all devices into a SPINE network. However, it should be noted that this is not required in general. In fact, in IP networks typical IP routing devices rather create individual IP sub-networks and do not easily expose the internal devices to the outside (hence also not to other sub-networks). Similarly, SPINE gateways and SPINE routers are not forced in general to expose SPINE devices from one of their interfaces to the other. This is finally implementation dependent unless further rules specify the exact behaviour. On one hand this permits to "shield" two or more SPINE networks from each other. On the other hand, it permits to "extend" networks (as shown with the example of a vendor specific technology gateway, e.g.).

Although the examples focused on router and technology gateway it should be recalled that "smart" devices may as well be used to forward messages. This finally depends on the implementation.

